UNIVERSIDADE FEDERAL DE VIÇOSA

Arquiteturas de aceleradores em hardware para o problema de posicionamento de grafos em CGRAs, FPGAs e QCAs

Jeronimo Costa Penha
Doctor Scientiae

VIÇOSA - MINAS GERAIS
2026

JERONIMO COSTA PENHA

Arquiteturas de aceleradores em hardware para o problema de posicionamento de grafos em CGRAs, FPGAs e QCAs

Tese apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Doctor Scientiae*.

Orientador: Ricardo dos S. Ferreira

Coorientador: Jose Augusto Miranda Nacif

VIÇOSA - MINAS GERAIS
2026

Ficha catalográfica elaborada pela Biblioteca Central da Universidade Federal de Viçosa - Campus Viçosa

T
P399f
2026

Penha, Jeronimo Costa, 1984-
Ferramentas para geração automática de aceleradores em plataformas heterogêneas de alto desempenho com FPGA / Jeronimo Costa Penha. – Viçosa, MG, 2026.
1 tese eletrônica (170 f.): il. (algumas color.).

Inclui apêndices.
Orientador: Ricardo dos Santos Ferreira.
Tese (doutorado) - Universidade Federal de Viçosa, Departamento de Informática, 2026.
Referências bibliográficas: f. 146-156.
DOI: https://doi.org/10.47328/ufvbbt.2026.416
Modo de acesso: World Wide Web.

1. Arquitetura de computador. 2. Algoritmo de grafos. 3. Linguagem de programação (Computadores). I. Ferreira, Ricardo dos Santos, 1969-. II. Universidade Federal de Viçosa. Departamento de Informática. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDD 22. ed. 006.66

Bibliotecário(a) responsável: Euzébio Luiz Pinto CRB-6/3317

JERONIMO COSTA PENHA

Arquiteturas de aceleradores em hardware para o problema de posicionamento de grafos em CGRAs, FPGAs e QCAs

Tese apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Doctor Scientiae*.

APROVADA: 5 de março de 2026.

Assentimento:

_____

Jeronimo Costa Penha
Autor

_____

Ricardo dos Santos Ferreira
Orientador

Dedico este trabalho aos meus pais, pelo amor, pelo apoio e por tudo aquilo que construíram em mim antes mesmo que eu soubesse o caminho que iria seguir.

À minha namorada, Kamila, pelo carinho, pela paciência, pela presença e pelo cuidado nos dias difíceis. Por me ouvir, me aconselhar, cuidar de mim quando precisei, dividir comigo o peso da rotina e tornar esta caminhada menos solitária.

E a todos que, de alguma forma, estiveram comigo nesta travessia.

AGRADECIMENTOS

Este trabalho foi realizado com o apoio das seguintes agências de pesquisa brasileiras: Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001, Fundação de Amparo à Pesquisa do Estado de Minas Gerais (FAPEMIG) e Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq). E ao Edital 001/2022 - Demanda Universal, Apq-01577-22 Projeto: “Mapeamento De Aceleradores De Alto Desempenho Em Arquiteturas De Domínio Específico”.

Veni, vidi, vici.

— Júlio César

RESUMO

PENHA, Jeronimo Costa, D.Sc., Universidade Federal de Viçosa, março de 2026. **Arquiteturas de aceleradores em hardware para o problema de posicionamento de grafos em CGRAs, FPGAs e QCAs**. Orientador: Ricardo dos Santos Ferreira. Coorientador: Jose Augusto Miranda Nacif.

O posicionamento de grafos é uma etapa fundamental no fluxo de projeto de diversas arquiteturas reconfiguráveis, como Field Programmable Gate Arrays (FPGAs), Coarse-Grained Reconfigurable Architectures (CGRAs) e arquiteturas emergentes baseadas em Quantum-dot Cellular Automata (QCA). Nesses sistemas, o posicionamento consiste em determinar a localização de operações, células lógicas ou elementos computacionais em uma matriz de processamento, de modo a reduzir o custo de comunicação entre os elementos e respeitar as restrições estruturais da arquitetura. Entretanto, trata-se de um problema computacionalmente complexo, classificado como NP-difícil, o que torna inviável a obtenção de soluções ótimas para grafos de grande porte por meio de busca exaustiva. Esta tese investiga a possibilidade de acelerar o processo de posicionamento por meio da implementação dedicada de algoritmos em hardware reconfigurável. Para isso, são estudados três algoritmos de posicionamento presentes na literatura: Simulated Annealing (SA) e os algoritmos baseados em travessia de grafos You Only Traverse Once (YOTO) e You Only Traverse Twice (YOTT). A partir desses algoritmos, propõe-se um modelo de aceleradores baseado em arquiteturas em pipeline, capaz de explorar paralelismo temporal e espacial em dispositivos FPGA para acelerar a busca por soluções de posicionamento. A abordagem proposta foi estruturada em três etapas principais. Na primeira, são investigadas as características do problema de posicionamento em diferentes arquiteturas reconfiguráveis, com destaque para as particularidades estruturais de CGRAs, FPGAs e circuitos baseados em QCA. Nessa etapa, são analisadas as restrições impostas por cada arquitetura e sua influência no processo de posicionamento. Na segunda etapa, são desenvolvidas arquiteturas de aceleradores para os algoritmos SA, YOTO e YOTT utilizando estruturas em pipeline. Essas arquiteturas exploram o paralelismo interno e o uso de múltiplos fluxos de execução para permitir a avaliação simultânea de diferentes soluções candidatas. Na terceira etapa, são realizados experimentos com conjuntos de benchmarks utilizados na literatura, incluindo MCNC, EPFL, MNIST e JSC. Os resultados obtidos permitem avaliar tanto a qualidade das soluções geradas quanto o desempenho das arquiteturas propostas. Os experimentos demonstram que a implementação dos algoritmos em

hardware possibilitou uma redução significativa no tempo de execução, mantendo qualidade de solução compatível com aquela obtida por implementações tradicionais em software o que representa uma abordagem promissora para reduzir o tempo de compilação de aplicações voltadas a arquiteturas reconfiguráveis. Como principal contribuição, esta tese apresenta um modelo genérico de aceleradores para algoritmos de posicionamento de grafos baseado em arquiteturas pipeline.

Palavras-chave: posicionamento de grafos; FPGA; arquiteturas reconfiguráveis; aceleração em hardware; arquiteturas em pipeline

ABSTRACT

PENHA, Jeronimo Costa, D.Sc., Universidade Federal de Viçosa, March, 2026. **Hardware accelerator architectures for the graph placement problem in CGRAs, FPGAs, and QCAs**. Adviser: Ricardo dos Santos Ferreira. Co-adviser: Jose Augusto Miranda Nacif.

Graph placement is a fundamental step in the design flow of several reconfigurable architectures, such as Field-Programmable Gate Arrays (FPGAs), Coarse-Grained Reconfigurable Architectures (CGRAs), and emerging architectures based on Quantum-dot Cellular Automata (QCA). In these systems, placement consists of determining the location of operations, logic cells, or computational elements in a processing array, in order to reduce the communication cost between elements while respecting the structural constraints of the architecture. However, this is a computationally complex problem, classified as NP-hard, which makes it infeasible to obtain optimal solutions for large graphs through exhaustive search. This thesis investigates the possibility of accelerating the placement process through dedicated implementations of algorithms in reconfigurable hardware. To this end, three placement algorithms found in the literature are studied: Simulated Annealing (SA) and the graph traversal-based algorithms You Only Traverse Once (YOTO) and You Only Traverse Twice (YOTT). Based on these algorithms, this work proposes an accelerator model based on pipeline architectures, capable of exploiting temporal and spatial parallelism in FPGA devices to accelerate the search for placement solutions. The proposed approach was structured into three main stages. In the first stage, the characteristics of the placement problem are investigated in different reconfigurable architectures, with emphasis on the structural particularities of CGRAs, FPGAs, and QCA-based circuits. In this stage, the constraints imposed by each architecture and their influence on the placement process are analyzed. In the second stage, accelerator architectures are developed for the SA, YOTO, and YOTT algorithms using pipeline structures. These architectures exploit internal parallelism and multiple execution flows to allow the simultaneous evaluation of different candidate solutions. In the third stage, experiments are carried out using benchmark sets commonly used in the literature, including MCNC, EPFL, MNIST, and JSC. The results obtained allow the evaluation of both the quality of the generated solutions and the performance of the proposed architectures. The experiments show that implementing the algorithms in hardware enabled a significant reduction in execution time while maintaining solution quality compatible with that obtained by traditional software implementations. This represents a

promising approach to reducing the compilation time of applications targeting reconfigurable architectures. As its main contribution, this thesis presents a generic accelerator model for graph placement algorithms based on pipeline architectures.

Keywords: graph placement; FPGA; reconfigurable architectures; hardware accelerationp; ipeline architectures

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

ABC *System for Sequential Synthesis and Verification*. 43, 98, 123, 125

ASIC *Application-Specific Integrated Circuits*. 43

B-RAM *Block RAM*. 5, 26, 43, 51, 53, 56, 58, 59, 69, 71, 72, 81, 82, 91, 95, 115–118, 143, 144

BFS *Breadth-First Search*. 1, 3, 37, 40, 76

BLIF *Berkeley Logic Interchange Format*. 98, 125, 168

CAD *Computer-Aided Design*. 25, 43, 139

CGRA *Coarse Grain Reconfigurable Architecture*. 6, 23, 25–27, 29, 30, 34, 41, 43, 46, 50–52, 57, 58, 60, 62–66, 69, 71, 73, 76, 77, 79, 81, 91, 95, 96, 119, 121, 125, 132–137, 143, 144

CMOS *Complementary metal-oxide-semiconductor*. 46, 47, 138

CPU *Central Processing Unit*. 80, 93, 123, 132, 133

DFG Data-Flow Graph. 41, 54, 57, 79, 85, 89, 95, 96, 133

DFS *Depth-First Search*. 1, 37, 40, 68, 69, 73, 76, 101, 105

DNN *Deep Neural Networks*. 44

DOT *DAG of tomorrow*. 98, 125

DSP *Digital Signal Processor*. 43, 44, 71, 81

FPGA *Field Programmable Gate Array*. 2, 3, 5, 6, 22–30, 34, 41, 43–45, 50–54, 56, 58–60, 62–65, 69, 71, 73, 76, 77, 79, 80, 83, 85, 91–95, 97, 98, 102, 108, 117, 119, 123, 125, 132–139, 143, 144

GCN *Graph Convolutional Networks*. 135, 137

GNN *Graph Neural Networks*. 32, 50, 135, 137

GPU *Graphics Processing Unit*. 22–24, 44, 133, 138, 139

HLS *High-Level Synthesis*. 5, 79, 85, 94, 95, 134

ILP *Integer Linear Programming*. 32, 50, 137

LFSR *Linear-feedback shift register*. 37

LTS *Long Term Support*. 124

LUT *Lookup Table*. 3–5, 30, 43, 44, 51, 53, 71–76, 81, 82, 91, 92, 94, 95, 97–101, 104–107, 109–112, 115, 116, 119–121, 123, 133, 139, 144, 159, 163

MAC *Multiply-Accumulate operation*. 44

ML *Machine Learning*. 136

NET *Packed Netlist Format*. 98

PC *Program Counter*. 66

PE *Processing Element*. 41, 50, 81, 132, 133

PLACE *Placement File Format*. 98, 170

QCA *Quantum-dot Cellular Automata*. 2, 3, 6, 23, 25–27, 29, 30, 46, 47, 50–52, 60–62, 64, 65, 73, 76–80, 119, 121, 124–127, 132, 136–138, 140–145

RAM *Random Access Memory*. 82, 117, 124

RISC *Reduced Instruction Set Computer*. 57

RL *Reinforcement Learning*. 135, 137, 138

RTL *Register Transfer Level*. 5, 79, 81, 94, 95, 143

SA *Simulated Annealing*. 1, 2, 4, 5, 26, 29, 30, 32–34, 36, 44, 50–52, 55–57, 60, 62, 63, 65–67, 69–72, 79, 80, 83–86, 88–97, 102–105, 107, 108, 110, 111, 124, 127–130, 133–139, 141–145

SAT Satisfatibilidade Booleana. 50, 134, 137

SIMD *Single Instruction, Multiple Data*. 22

SMT *Simultaneous Multithreading*. 5, 66, 91, 140, 142

SMT Solver *Satisfiability Modulo Theory*. 51

URAM *Ultra RAM Block*. 117, 118

USE *Universal, Scalable and Efficient*. 2, 47, 48, 60–62, 76, 80, 125, 126, 140–142, 145

VPR *Versatile Place and Route*. 3, 5, 43, 44, 73, 79, 80, 83–85, 92, 97, 98, 101, 103–111, 118, 121, 122, 124, 135–137

VTR *Verilog-to-Routing*. 43, 44, 134

XML *Extensible Markup Language*. 97, 98

YOTO *You Only Traverse Once*. 1–3, 5, 26, 29, 30, 32, 34–40, 51, 65–70, 72, 73, 76, 77, 79, 85–87, 89–93, 95–97, 100–103, 105, 107, 108, 110–112, 114–116, 118, 119, 121, 122, 136, 137, 139, 142–145

YOTT You Only Traverse Twice. 2–5, 26, 29, 30, 32, 34, 37, 38, 40, 51, 65, 68–70, 72, 73, 76–79, 85, 86, 89–93, 95–97, 100–103, 105, 107, 108, 110–112, 118, 130, 131, 136, 137, 139, 142–145

ZBDD *Zero-suppressed Binary Decision Diagram*. 134, 137

SUMÁRIO

Capítulo 1

Introdução

A crescente demanda por desempenho computacional, associada a restrições de consumo energético e dissipação térmica, tem impulsionado o desenvolvimento de arquiteturas especializadas ao longo das últimas décadas. Nesse contexto, aceleradores de domínio específico emergiram como uma alternativa para superar as limitações das arquiteturas de propósito geral, uma vez que possibilitam a obtenção de ganhos significativos em relação ao desempenho por watt [23]. Entre essas tecnologias, os FPGAs destacam-se pela capacidade de adaptação às características das aplicações alvo.

Entretanto, apesar de seu potencial, a adoção prática de aceleradores baseados em FPGAs permanece limitada em virtude do elevado tempo requerido nas etapas de síntese, especialmente no posicionamento e roteamento. Mesmo para circuitos regulares e modularizados, essas etapas podem exigir horas ou até dias para a geração do *bitstream* final, constituindo um gargalo significativo em fluxos de projeto modernos [69]. Essa limitação compromete aplicações que exigem rápida adaptação do *hardware*, como em cenários de computação em borda.

O problema do posicionamento e do roteamento caracteriza-se por decisões interdependentes, acesso irregular a estruturas de dados e dependência global do estado do sistema. Essas propriedades dificultam sua aceleração em arquiteturas paralelas convencionais. Embora GPUs ofereçam elevado paralelismo computacional, sua organização baseada em um modelo de execução SIMD e controle de fluxo compartilhado a torna pouco adequada para problemas de paralelismo irregular, como o posicionamento de grafos, nos quais diferentes instâncias de execução apresentam cargas computacionais desbalanceadas e caminhos de decisão distintos [43, 13].

Por outro lado, os FPGAs oferecem um modelo de computação distinto, baseado em paralelismo espacial e controle personalizado do fluxo de dados, o que os torna adequados para algoritmos com estruturas irregulares [34]. No entanto, essa flexibilidade vem acompanhada de desafios, como a complexidade no desenvolvimento, a dependência de conhecimento especializado em arquitetura de *hardware* e, sobretudo, os longos tempos de compilação associados às etapas tradicionais de posicionamento e roteamento [22].

Diante desse cenário, esta tese fundamenta-se na hipótese de que o gargalo imposto pelo tempo de posicionamento pode ser mitigado por meio da aceleração em *hardware* do próprio processo de posicionamento. Dessa forma, propõe-se o desenvolvimento de aceleradores especializados, projetados para algoritmos iterativos e estocásticos de posicionamento.

A abordagem adotada nesta tese baseia-se na implementação em *hardware* de algoritmos de posicionamento baseados em travessia de grafos e em *Simulated Annealing*, estruturados em arquiteturas *pipeline* e com uso de memória interna e distribuída. A proposta visa a exploração do paralelismo espacial e temporal do hardware reconfigurável, buscando a execução dessas etapas com latências da ordem de nanossegundos, de acordo com o que foi observado em estudos preliminares [24].

Além do posicionamento em FPGAs, a tese investiga a aplicação dos aceleradores em outras arquiteturas, como CGRAs, que operam em granularidade mais grossa e apresentam desafios adicionais de balanceamento temporal [58, 65] e QCAs, as quais possuem restrições adicionais no nível de interconexões e de sincronização [79].

1.1 Motivação

Embora arquiteturas paralelas de propósito geral, como GPUs, tenham sucesso em aplicações caracterizadas por paralelismo regular, sua eficiência é reduzida em problemas que envolvem paralelismo irregular. Em particular, a execução em GPUs é estruturada em grupos de *threads* denominados *warps*, que compartilham um único fluxo de controle. Dessa forma, o tempo de execução de um *warp* é determinado pela *thread* mais lenta, o que penaliza algoritmos iterativos nos quais o número de operações varia entre instâncias.

Por outro lado, os FPGAs oferecem uma estrutura capaz de explorar o paralelismo espacial, no qual múltiplas unidades de processamento independentes podem ser instanciadas e operar de forma assíncrona. Essa característica permite que tarefas com durações distintas sejam executadas sem a necessidade de sincronização rígida, o que favorece a exploração de paralelismo irregular [34]. Estudos preliminares desenvolvidos no início deste trabalho demonstraram esse potencial por meio da implementação de aceleradores para algoritmos iterativos baseados em grafos em FPGAs, em comparação com abordagens em GPUs [24].

Observou-se que a replicação de unidades de processamento independentes em FPGAs permite maximizar a utilização dos recursos disponíveis, mesmo quando diferentes instâncias demandam números distintos de iterações. Um exemplo disso é ilustrado na Figura 1.1, que apresenta uma arquitetura com múltiplos elementos de processamento conectados por meio de multiplexadores (Figura 1.1(a)) e uma alternativa baseada em interconexão em cadeia (Figura 1.1(b)). Enquanto a primeira abordagem

introduz contenções e um maior custo em área, a segunda reduz a complexidade da interconexão, o que permite uma maior escalabilidade do número de unidades de processamento [96, 24].

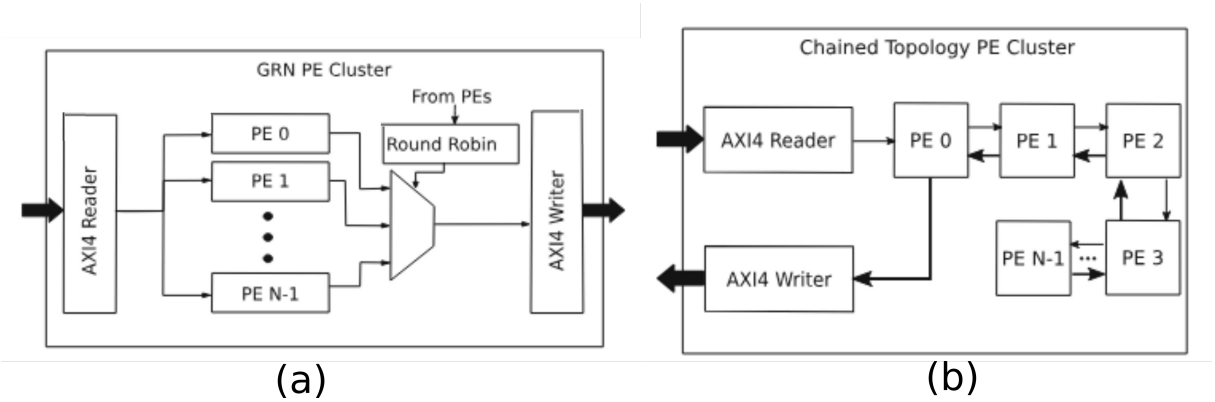


Figura 1.1: (a) Múltiplos elementos de processamento e conexão com multiplexadores; (b) Múltiplos elementos interligados em cadeia.

Apesar dessas vantagens, a adoção prática de aceleradores baseados em FPGAs enfrenta desafios relevantes. Entre eles, destaca-se o elevado custo em área associado a estruturas de interconexão complexas, como multiplexadores de grande porte, bem como o longo tempo requerido pelas etapas de síntese. Mesmo quando soluções eficientes são obtidas para a execução em tempo real, o tempo de compilação pode superar em ordens de magnitude o tempo de execução da aplicação, o que compromete a viabilidade [24, 69].

Além dos desafios associados à interconexão e ao tempo de síntese, arquiteturas paralelas baseadas em GPUs também apresentam limitações na execução de códigos com grandes estruturas condicionais. A avaliação de árvores de decisão, por exemplo, resulta em múltiplas instruções condicionais aninhadas, o que leva à divergência de execução entre *threads* de um mesmo *warp* e, consequentemente, à perda de desempenho [66]. A Figura 1.2 ilustra uma árvore de decisão binária, sua codificação tradicional com condicionais e uma versão equivalente baseada em operações aritméticas, proposta como alternativa para a mitigação desse problema em GPUs.

Embora abordagens baseadas na transformação de condicionais em expressões aritméticas possam melhorar o desempenho em determinados cenários, sua complexidade cresce exponencialmente com a profundidade da árvore e limita sua utilização prática [66]. Em contraponto, os FPGAs permitem a implementação dessas estruturas por meio de multiplexadores ou memórias indexadas, além da possibilidade de exploração de *pipeline* para aumento de desempenho [62].

Essas limitações motivam a investigação de novas abordagens para a redução do tempo das etapas de posicionamento e roteamento, que constituem gargalos centrais no uso de FPGAs. Em vez de recorrer a arquiteturas complexas baseadas em máquinas

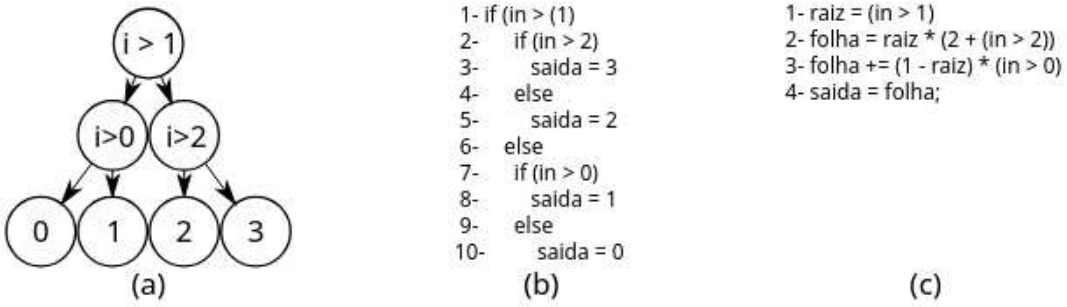


Figura 1.2: (a) Exemplo de uma simples árvore de decisão; (b) Mapeamento usando *if* aninhados; (c) Código equivalente sem uso de condicionais.

de estados sequenciais ou soluções fortemente acopladas a fluxos tradicionais de CAD, esta tese propõe a exploração de aceleradores especializados, projetados para executar algoritmos de posicionamento de forma paralela, escalável e com baixa complexidade estrutural.

Adicionalmente, a motivação deste trabalho se estende a domínios semelhantes. Arquiteturas do tipo CGRA trazem desafios adicionais relacionados ao balanceamento temporal dos fluxos de dados [58, 65], enquanto tecnologias emergentes, como QCAs, impõem restrições de interconexão e de sincronização semelhantes às observadas em arquiteturas reconfiguráveis [79].

1.2 O Problema

Um dos principais desafios no desenvolvimento de aceleradores em FPGAs consiste no elevado tempo necessário nas etapas de posicionamento e roteamento. Uma estratégia para mitigar esse gargalo baseia-se no aumento da granularidade do *hardware* reconfigurável, por meio da adoção de arquiteturas do tipo CGRA (*Coarse-Grained Reconfigurable Architecture*) [58], que operam em nível de palavra, em contraste com o nível de bit característico dos FPGAs. Entretanto, o mapeamento em CGRAs introduz restrições adicionais, em especial a necessidade de balanceamento dos caminhos de dados, o que exige a resolução conjunta dos problemas de posicionamento, roteamento e escalonamento para assegurar o funcionamento correto [65].

Trabalhos recentes investigam o uso de aceleradores implementados em FPGAs para auxiliar o mapeamento em CGRAs [82]. Contudo, essas abordagens apresentam limitações relacionadas à subutilização de recursos, decorrentes da natureza sequencial das máquinas de estados finitos empregadas na modelagem do fluxo de operações [82]. Nesse contexto, esta tese propõe uma nova abordagem para o problema de posicionamento em CGRAs, de modo a explorar as características da arquitetura dos FPGAs

e a utilização de estruturas flexíveis capazes de suportar diferentes algoritmos, com ênfase em métodos baseados em travessia de grafos e em *Simulated Annealing* (SA).

A proposta estende ainda o uso de aceleradores de posicionamento para domínios correlatos, como o posicionamento em tecnologias emergentes baseadas em QCAs (*Quantum-dot Cellular Automata*) [79], que compartilham similaridades conceituais com arquiteturas CGRA, especialmente no que se refere às restrições topológicas e à necessidade de sincronização. Apesar das diferenças físicas entre as tecnologias, essa convergência indica a viabilidade da reutilização de modelos de aceleradores de posicionamento em múltiplos contextos arquiteturais.

Por fim, coloca-se o desafio do posicionamento no próprio FPGA. De forma análoga ao uso de compiladores para a construção de novos compiladores, questiona-se se um acelerador de posicionamento implementado em FPGA pode ser empregado para acelerar o processo de posicionamento de circuitos em FPGA, ampliando a eficiência do próprio fluxo de projeto.

1.3 Objetivo

O objetivo geral desta tese é o desenvolvimento de metodologias e arquiteturas para a implementação de aceleradores em hardware usando FPGA como tecnologia alvo. O foco concentra-se em um nicho específico de aplicações relacionadas a problemas de posicionamento. A proposta consiste no desenvolvimento de uma arquitetura específica para esse domínio, com flexibilidade suficiente para abranger uma classe de problemas de posicionamento aplicáveis a três tecnologias distintas: CGRAs, nanotecnologia baseada em QCA e a própria arquitetura de FPGAs. A flexibilidade da proposta é avaliada por meio da implementação de três algoritmos distintos de posicionamento.

Outro ponto importante é a exploração da capacidade intrínseca do FPGA para suporte à computação espacial e temporal, na qual serão exploradas arquiteturas em *pipeline* e a execução de múltiplas tarefas irregulares de forma concorrente, unidas às características dos FPGAs atuais, como memórias B-RAM distribuídas.

Os objetivos específicos são a implementação dos algoritmos de posicionamento baseados em travessia simples (YOTO) [31, 33], travessia dupla (YOTT) [15] e SA [17] em *hardware* com arquitetura em *pipeline* e memória distribuída, usando módulos B-RAM do FPGA. Conforme mencionado anteriormente, os três algoritmos implementados em hardware serão avaliados em três cenários com restrições distintas. No contexto dos grafos de CGRA, estes são gerados por compiladores, onde cada vértice é uma instrução ou operação em nível de palavra; enquanto nos dois outros cenários, nanotecnologia e FPGA, os vértices dos grafos são funções lógicas. Outro ponto específico é explorar diferentes estratégias de travessia do grafo (largura, profundidade e *zigzag*).

1.4 Contribuições

A principal contribuição é um modelo genérico de aceleradores para algoritmos iterativos e estocásticos de posicionamento, ilustrado na Figura 1.3 (a). Suponha um algoritmo iterativo que melhore a solução a cada iteração ou que posicione um vértice a cada iteração. O algoritmo pode ser composto por uma sequência de passos com dependência, onde o passo i depende da finalização do passo $i-1$. Além disso, o passo i pode fazer acessos a uma memória local ou global. Para explorar o paralelismo temporal da arquitetura, os passos são executados em *pipeline*. Como o algoritmo é estocástico, pode-se executar várias instâncias do mesmo em *pipeline*, similar a uma execução *multithread*, como ilustrado na Figura 1.3 (b). Esta abordagem permite n execuções com compartilhamento de recursos e esconde a latência de cada execução com o paralelismo temporal.

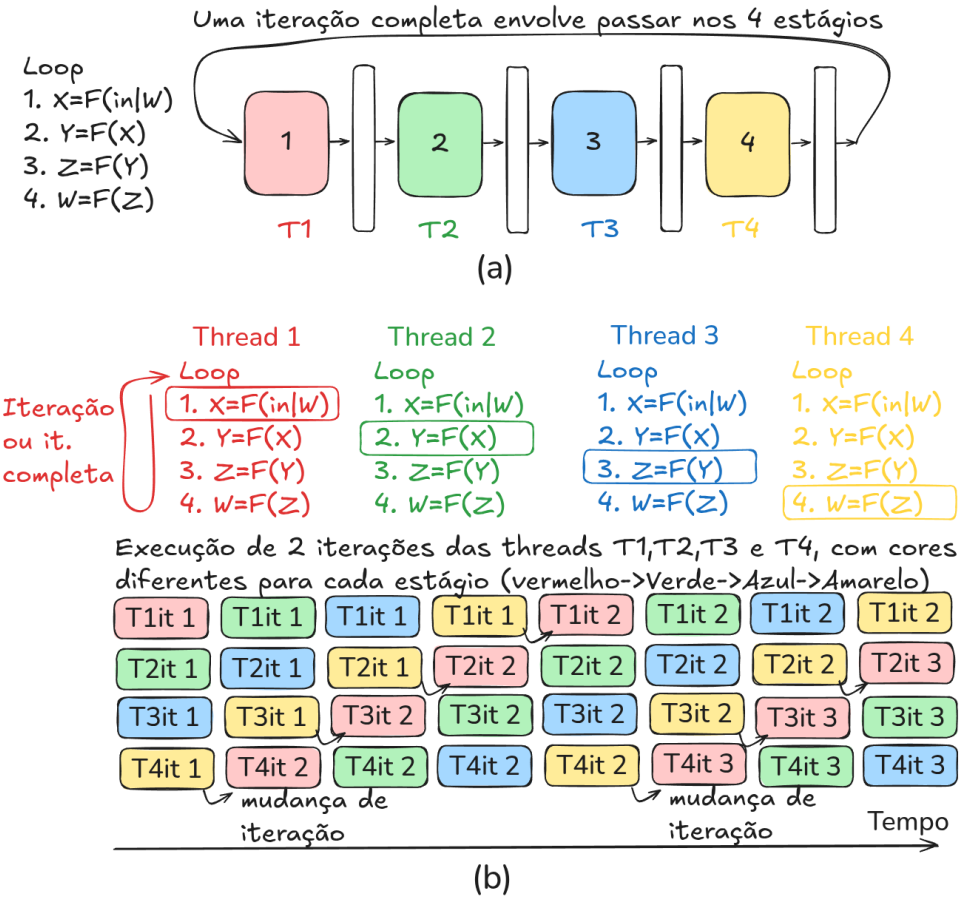


Figura 1.3: (a) Arquitetura *Pipeline* genérica com exemplo com quatro estágios; (b) Paralelismo Temporal com 4 instâncias de execução em estágio diferentes representadas por 4 *threads*.

Outra contribuição consiste na demonstração dos ganhos de desempenho obtidos com a implementação de aceleradores em *hardware* dedicados à resolução do problema de posicionamento em arquiteturas distintas, abrangendo CGRAs, FPGAs e QCAs.

Adicionalmente, introduz-se o conceito de *pipeline* vertical, concebido como uma estratégia para a redução da demanda por recursos de roteamento interno. Essa abordagem baseia-se na utilização eficiente das filas já disponíveis nos dispositivos FPGAs, que permitem o encadeamento de estágios computacionais com menor sobrecarga de interconexão. Complementarmente, propõe-se o uso de múltiplas memórias *cache* distribuídas e independentes, com o objetivo de avaliar seu impacto na escalabilidade das arquiteturas implementadas. Essa estratégia viabiliza tanto a manipulação de grafos maiores quanto a otimização do uso de área, o que possibilita a replicação de instâncias do acelerador e, consequentemente, a ampliação do grau de paralelismo e do ganho de aceleração.

1.5 Publicações e Submissões

Em termos de publicação dos resultados parciais deste trabalho, um artigo foi publicado a nível nacional no WSCAD 2023 e outro artigo está pronto para submissão ao IEEE Transactions on CAD:

- Penha, Jeronimo Costa et al. *Simulated Annealing* em *Hardware* com Múltiplas *Threads* em *Pipeline* para Posicionamento em CGRAs. Em: Anais do XXIII Simpósio em Sistemas Computacionais de Alto Desempenho. SBC, 2022. p. 97-108 **(Publicado)**.

- Penha, Jeronimo Costa et al. *Domain-specific Stochastic Acceleration by exploiting SMT, FPGA Pipelines, and Block RAMs* **(A submeter)**.

- Penha, Jeronimo Costa et al. *Hardware-Accelerated Runtime Placement Engine for Low-Latency Machine Learning FPGA* **(A submeter)**.

Como mencionado na seção 1.1, duas implementações que exploram aspectos específicos das GPUs e dos FPGAs foram desenvolvidas no início deste trabalho:

- Penha, Jeronimo et al. Avaliação de Estilos de Código para Árvores de Decisão em GPU com *Microbenchmarks*. In: Anais do XXIV Simpósio em Sistemas Computacionais de Alto Desempenho. SBC, 2023. p. 277-288.

- Penha, Jeronimo et al. *Gene regulatory accelerators on cloud FPGA*. Concurrency and Computation: Practice and Experience, v. 35, n. 24, p. e7822, 2023.

Os resultados parciais do posicionamento para FPGA estão em fase de redação para a submissão de um artigo em um periódico ou conferência de destaque na área (DATE, DAC, FPGA, FCCM).

1.6 Organização do texto

Esta tese está organizada da seguinte forma. O Capítulo 2 apresenta os algoritmos de posicionamento da literatura selecionados para implementação em *hardware*. O Capítulo 3 descreve a arquitetura proposta e sua implementação em *hardware* para os algoritmos SA, YOTO e YOTT, considerando os problemas de posicionamento em CGRA, FPGA e QCA. O Capítulo 4 apresenta os experimentos realizados e os resultados obtidos nos três domínios avaliados. O Capítulo 5 reúne a revisão bibliográfica, destacando os principais trabalhos relacionados em cada área. Por fim, o Capítulo 6 apresenta as conclusões desta tese.

Capítulo 2

Fundamentos Teóricos

Este capítulo apresenta o problema de posicionamento e os três algoritmos considerados para implementação em *hardware*, avaliados em três contextos arquiteturais distintos: CGRA, FPGA e QCA. A Seção 2.1 introduz o problema de posicionamento em matrizes bidimensionais e discute sua complexidade computacional. Em seguida, são apresentados três algoritmos da literatura [31, 15, 17], selecionados para implementação em *hardware* neste trabalho.

A partir da adoção de três arquiteturas-alvo, o capítulo organiza-se da seguinte forma: Inicialmente, os três algoritmos são apresentados no contexto das arquiteturas CGRA. Em seguida, são discutidas as adaptações e as diferenças necessárias para as implementações em FPGA e QCA.

Além das distinções entre as arquiteturas, os grafos de entrada considerados em cada caso também diferem entre si. Para os CGRAs, o grafo de entrada corresponde a um grafo de fluxo de dados em nível de instruções. Em contrapartida, nas arquiteturas FPGA e QCA, o grafo de entrada representa um circuito combinacional. Especificamente, para FPGAs, o circuito é modelado como um grafo de LUTs, enquanto para QCA, o grafo é composto por portas lógicas de duas entradas.

A Seção 2.2 descreve o algoritmo *Simulated Annealing* (SA) aplicado à arquitetura CGRA, com destaque para sua estrutura iterativa baseada em trocas estocásticas. Em seguida, a Seção 2.3 apresenta os algoritmos de travessia, compostos pelo algoritmo de travessia simples (YOTO), detalhado na Seção 2.3.1, e pelo algoritmo de travessia dupla (YOTT), apresentado na Seção 2.3.2. Diferentemente do SA, os algoritmos de travessia constroem o posicionamento de forma incremental e alocam um novo vértice a cada iteração, até que todo o grafo seja percorrido. Esses algoritmos dependem, portanto, da estratégia de travessia adotada, que pode envolver busca em largura, busca em profundidade ou o método *zigzag*. Este último é detalhado na Seção 2.4 e foi proposto em [16].

No que se refere às arquiteturas, as Seções 2.5, 2.6 e 2.7 apresentam as arquiteturas CGRAs (*Coarse-Grained Reconfigurable Architectures*), FPGAs (*Field-Programmable Gate Arrays*) e QCAs (*Quantum-dot Cellular Automata*), respectivamente. Para cada uma delas, são discutidas as principais características, bem como as semelhanças e diferenças

que impactam diretamente as estratégias de posicionamento e as implementações em *hardware*.

2.1 Posicionamento de Grafos em Matrizes

O problema de posicionamento de grafos em matrizes é classificado como NP-difícil [41]. Para fins ilustrativos, considera-se inicialmente um grafo composto por nove vértices, apresentado na Figura 2.1 (a). A tarefa de posicionamento consiste na alocação desses vértices em uma matriz bidimensional, conforme exemplificado na Figura 2.1 (b).

Neste exemplo, adota-se como métrica de custo a distância *Manhattan*, definida como a soma das diferenças absolutas entre as coordenadas de linha e coluna. O objetivo do problema consiste em minimizar o custo global do posicionamento, definido como a soma das distâncias associadas às arestas do grafo que conectam os vértices. Para cada aresta dirigida $a \rightarrow b$, os vértices a e b devem ser alocados, sempre que possível, em células adjacentes ou espacialmente próximas, de modo a reduzir o custo individual da aresta. Se a conexão entre duas células não ocorre de forma direta, o custo da aresta corresponde à soma dos comprimentos dos segmentos de interconexão necessários. O objetivo final é minimizar o custo global do posicionamento que corresponde à soma dos custos de todas as arestas.

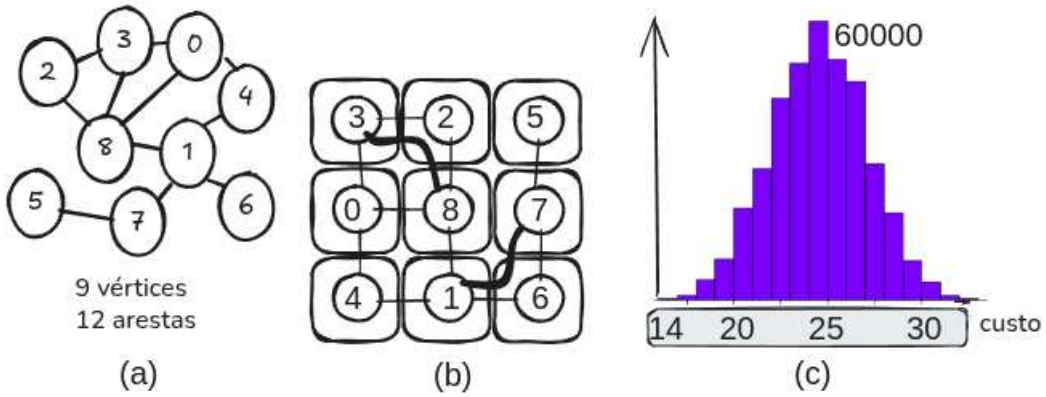


Figura 2.1: (a) Grafo de exemplo; (b) Mapeamento na matriz; (c) Histograma de custo para o mapeamento.

Na Figura 2.1 (b), a distância total associada ao posicionamento do grafo é igual a 14. Para grafos de pequena dimensão, é possível explorar exaustivamente todo o espaço de soluções. A Figura 2.1 (c) apresenta o histograma dos custos obtidos para todos os 9! (isto é, 362.880) posicionamentos possíveis. Observa-se que aproximadamente 60.000 configurações, correspondentes a cerca de 16% do total, resultam em um custo igual a 25, valor significativamente distante da solução ótima. O custo máximo observado

foi de 33, enquanto o menor custo foi igual a 14. Apenas 8 das 362.880 configurações correspondem a soluções ótimas, o que representa uma probabilidade inferior a $0,002\%$ de obtenção da solução ótima por meio de uma seleção aleatória.

Dessa forma, a configuração apresentada na Figura 2.1 (b) caracteriza-se como uma solução ótima, ainda que algumas arestas, como $3 \rightarrow 8$ e $1 \rightarrow 7$, apresentem distância igual a 2 na matriz. Essas arestas são destacadas visualmente na figura por meio de linhas em negrito. Nessa solução, todas as arestas são implementadas com o menor número possível de segmentos de conexão e apenas duas delas requerem dois segmentos. Consequentemente, o custo total mínimo do posicionamento corresponde ao número total de arestas acrescido de dois segmentos adicionais. Embora simples, esse exemplo mostra que, mesmo em soluções ótimas, nem sempre é possível posicionar todos os vértices em células adjacentes.

A abordagem por força bruta, embora viável para grafos de pequena escala, torna-se impraticável à medida que o número de vértices aumenta. Por exemplo, um grafo com 25 vértices possui um espaço de busca composto por 25! posicionamentos possíveis, enquanto um grafo com 64 vértices gera 64! combinações, número que supera a estimativa do total de átomos existentes no universo observável.

Diante dessa limitação, diversas heurísticas têm sido propostas na literatura. Entre elas, destacam-se abordagens híbridas [65] que combinam programação inteira linear (ILP), heurísticas gulosas, algoritmos genéticos [25], técnicas de aprendizado por reforço [54] e redes neurais baseadas em grafos (GNN) [53]. No contexto desta tese, optou-se pela adoção do algoritmo SA [17] e de duas abordagens fundamentadas em travessias de grafos: os algoritmos YOTO [31] e YOTT [15], em razão de sua simplicidade estrutural e desempenho computacional.

2.2 O Algoritmo *Simulated Annealing*

O algoritmo *Simulated Annealing* (SA) constitui uma das abordagens mais eficazes para o problema de posicionamento de grafos [63, 90]. Esse desempenho, entretanto, apresenta como contrapartida uma alta demanda computacional em termos de tempo de execução. O algoritmo parte de uma solução inicial gerada de forma aleatória [64] e, ao longo de sua execução, realiza um grande número de operações de troca de posição (*swaps*) entre pares de células, que podem alcançar a ordem de milhões.

A Figura 2.2 (a) apresenta um exemplo de grafo, e a Figura 2.2 (b) mostra um posicionamento inicial aleatório dos nós sobre a matriz de posicionamento. Observa-se que os nós e e d, que são adjacentes ao nó a, foram alocados inicialmente em posições afastadas. A simples troca dos nós a e b nas respectivas células resulta em um posicionamento mais próximo de a em relação aos seus vizinhos d e e que reduz o custo associado a essas conexões.

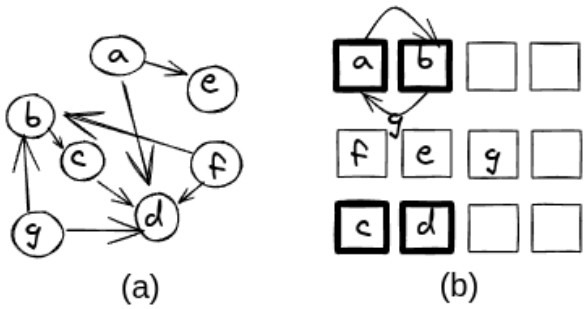


Figura 2.2: (a) Grafo; Arquitetura: (b) $a \leftrightarrow b$; (c) $c \leftrightarrow d$

A Figura 2.3 (a) apresenta o pseudocódigo do algoritmo SA. O algoritmo possui natureza iterativa e é estruturado em um laço composto por uma sequência de etapas interdependentes. Inicialmente, duas células, identificadas por C_a e C_b, são selecionadas aleatoriamente. Em seguida, os nós alocados nessas células, a e b, são identificados por meio de uma estrutura de dados responsável pelo mapeamento de células para nós (C_2N), a qual pode ser implementada como uma memória simples.

Na etapa subsequente, o grafo é consultado com o objetivo de obter os conjuntos de nós adjacentes a a e b, identificados por V_a e V_b, respectivamente. Para essa finalidade, o grafo pode ser armazenado em uma memória indexada pelo identificador do vértice, que permite o acesso direto às listas de arestas associadas.

Posteriormente, determina-se a posição, na matriz, dos nós pertencentes aos conjuntos V_a e V_b, o que resulta na formação dos conjuntos de células VC_a e VC_b. Essa operação utiliza a estrutura de mapeamento inverso, de nó para célula (N_2C), que também pode ser implementada por meio de uma memória dedicada. Com essas informações, calcula-se o custo associado às arestas incidentes aos nós a e b, com base na distância de *Manhattan* entre os pares de nós antes e após a realização da troca.

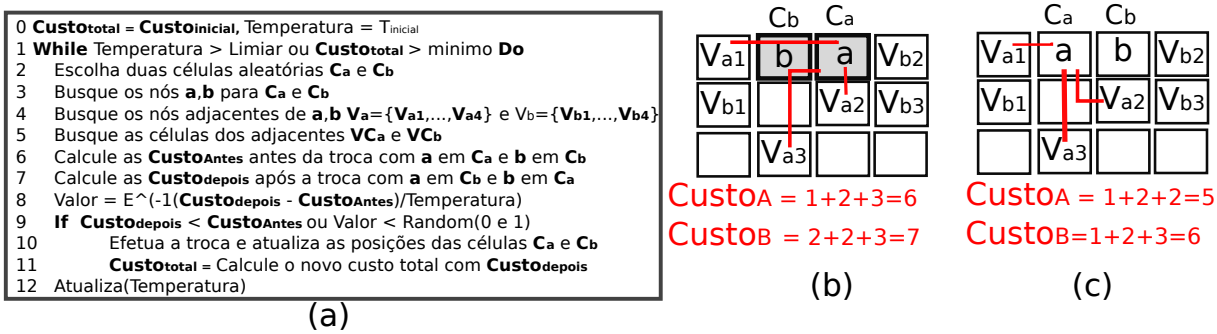


Figura 2.3: (a) Pseudo-Código do SA; (b) Antes $C_a \leftrightarrow C_b$; (c) Depois da troca.

A Figura 2.3 (b) exemplifica o cálculo do custo associado às distâncias entre o nó a e seus nós adjacentes. Nesse exemplo, o nó V_{a1} encontra-se a uma distância igual a 2 em relação a a, enquanto os nós V_{a2} e V_{a3} apresentam distâncias iguais a 1 e 3, respectivamente, o que resulta em um custo total igual a 6. Para o nó b, a

soma das distâncias em relação aos seus nós adjacentes é igual a 7. Caso os nós *a* e *b* sejam permutados, conforme ilustrado na Figura 2.3 (c), observa-se uma redução nas distâncias em relação aos respectivos vizinhos, o que justifica a aceitação da troca pelo algoritmo.

Desafios para a implementação do SA em *hardware*

A implementação do algoritmo SA em *hardware* apresenta desafios significativos associados à sua natureza sequencial, decorrente das dependências de dados entre as operações sucessivas do laço principal. A cada iteração, duas células candidatas à troca são selecionadas, e a decisão de aceitar ou rejeitar essa troca depende de uma sequência de operações interdependentes, o que limita a exploração do paralelismo em nível de instrução ou de dados.

Além disso, o algoritmo requer acessos frequentes a estruturas de dados em memória responsáveis pelo mapeamento entre os nós do grafo e suas posições na matriz de posicionamento (N_2C), bem como ao mapeamento inverso (C_2N). Essas estruturas devem ser atualizadas a cada operação de troca, o que impõe restrições adicionais na prevenção de conflitos, a fim de garantir acessos concorrentes a memória para buscar a exploração de um paralelismo ao longo da execução.

Outro desafio para a implementação do algoritmo SA refere-se ao cálculo do custo associado a cada possível troca de posição entre dois nós. Esse cálculo exige múltiplos acessos às estruturas de memória que armazenam o mapeamento do grafo e o das células da arquitetura. Além disso, cada iteração do algoritmo depende diretamente do estado da iteração anterior.

2.3 Algoritmos de Travessia

Algoritmos de posicionamento baseados em travessia de grafos têm se mostrado alternativas promissoras para arquiteturas CGRA [48, 20, 15] e FPGAs [30, 32]. Esta seção apresenta a descrição detalhada dos algoritmos YOTO [31] e YOTT [15] para CGRA, abordados, respectivamente, nas Seções 2.3.1 e 2.3.2.

2.3.1 O Algoritmo de Travessia Simples (YOTO)

O algoritmo YOTO é uma abordagem gulosa que realiza, de forma concomitante, a travessia do grafo de dados a ser posicionado e do grafo-alvo que representa a matriz de posicionamento. Em virtude de sua natureza heurística, o YOTO não assegura a visita de todas as arestas do grafo, o que pode impactar negativamente a qualidade do roteamento final, em particular no caso das conexões não exploradas durante a travessia.

Por outro lado, essa simplificação confere ao algoritmo complexidade polinomial, uma vez que, a cada iteração, um único vértice do grafo é posicionado. Assim, o custo computacional do YOTO é da ordem de $O(N)$, em que N denota o número total de vértices do grafo [70, 40].

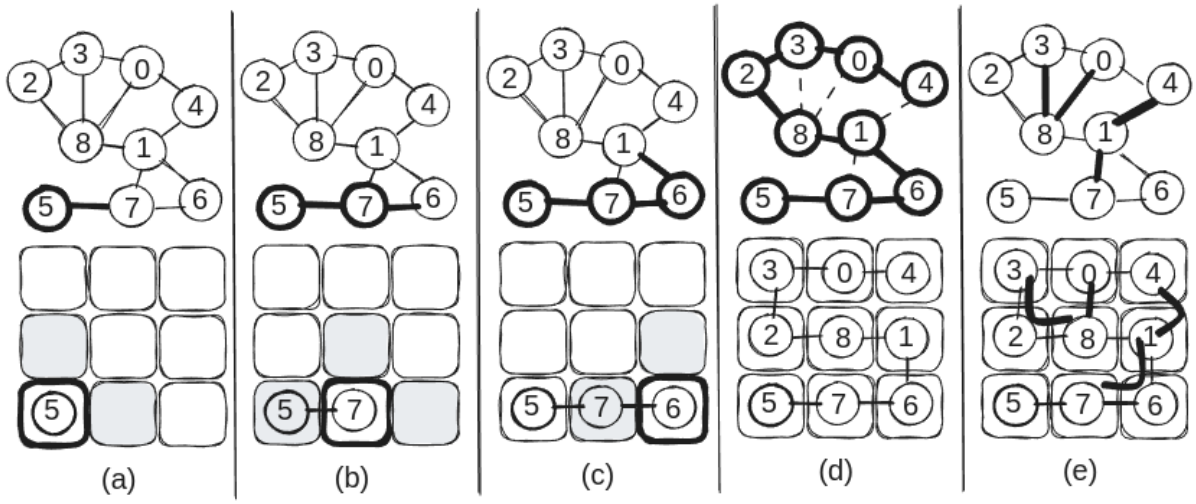


Figura 2.4: (a) Primeira aresta; (b) Segunda aresta; (c) Terceira aresta; (d) Última aresta; (e) Arestas não visitadas.

A Figura 2.4 (a) ilustra um exemplo do início da execução do algoritmo YOTO. A partir de um vértice arbitrário, neste caso, o vértice 5, o algoritmo realiza sua alocação inicial em uma célula escolhida aleatoriamente na matriz de posicionamento. A partir desse ponto, a cada iteração, uma aresta do vértice 5 é selecionada e irá orientar o posicionamento do vértice adjacente correspondente.

Na primeira iteração, a aresta $5 \rightarrow 7$ é processada. Como o vértice 5 já se encontra alocado, o vértice 7 é posicionado em uma célula adjacente a célula onde o vértice 5 foi alocado que esteja disponível na matriz. A Figura 2.4 (a) apresenta duas opções viáveis (norte e leste) e, para fins ilustrativos, assume-se a seleção aleatória da célula a leste, conforme mostrado na Figura 2.4 (b).

Na iteração seguinte, a aresta $7 \rightarrow 6$ é selecionada. Nesse estágio, existem três células adjacentes possíveis para a alocação do vértice 6; entretanto, uma delas já está ocupada pelo vértice 5, o que restringe o espaço de posicionamento. O algoritmo, então, seleciona aleatoriamente uma das posições livres. No exemplo, foi escolhida novamente a célula a leste.

Vale observar que as decisões aleatórias tomadas durante a travessia tem impacto no posicionamento final. O vértice 7 possui dois vizinhos, 1 e 6; contudo, a escolha da aresta $7 \rightarrow 6$ implica que a aresta $7 \rightarrow 1$ não será explorada ao longo desse percurso específico.

O posicionamento final obtido para esse exemplo, apresentado na Figura 2.4 (d), resulta da alocação dos vértices com base em 12 das 14 arestas originais do grafo.

Arestas como $7 \rightarrow 1$, $0 \rightarrow 2$, $8 \rightarrow 0$ e $4 \rightarrow 1$ não são visitadas durante a travessia, o que pode comprometer a proximidade espacial entre os vértices correspondentes. Ainda assim, conforme indicado na Figura 2.4 (e), parte dessas conexões não exploradas apresenta distâncias aceitáveis, com duas arestas posicionadas em células adjacentes e duas com distância igual a 2.

Esse exemplo resulta em uma solução ótima com custo total igual a 14, obtida em apenas 8 iterações, o que evidencia o potencial de eficiência do YOTO quando comparado ao SA.

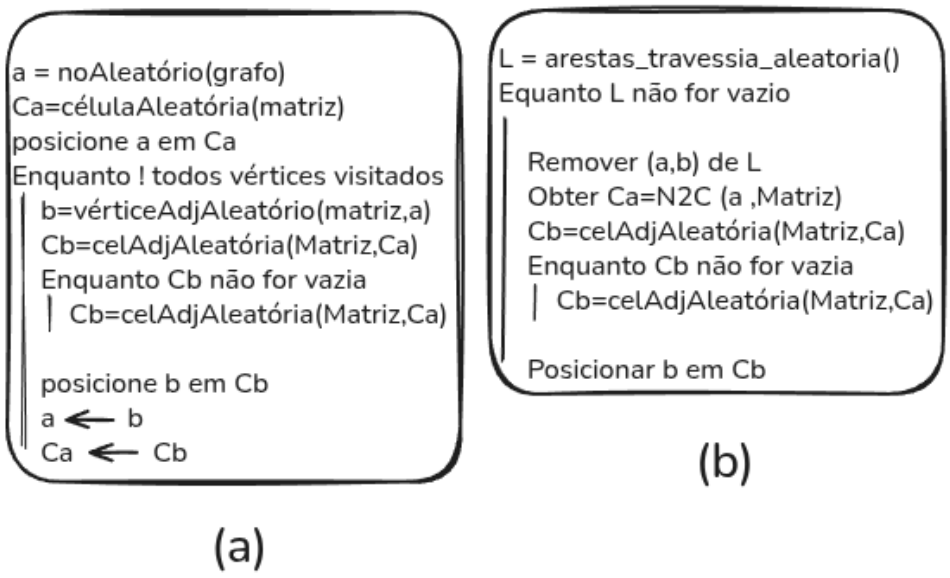


Figura 2.5: (a) Pseudo código para o YOTO; (b) Pseudo código baseado em listas para o YOTO.

A Figura 2.5 (a) apresenta o pseudocódigo do algoritmo YOTO. Em cada iteração, uma aresta $a \rightarrow b$ é selecionada e o vértice a deve estar alocado em uma célula C_a da matriz de posicionamento. O objetivo consiste em posicionar o vértice b em uma célula C_b, preferencialmente adjacente a C_a, de modo a minimizar a distância associada à aresta considerada.

Inicialmente, um vértice arbitrário é selecionado como ponto de partida e alocado aleatoriamente em uma célula da matriz. A partir desse posicionamento inicial, o laço principal do algoritmo prossegue com a seleção das arestas e o posicionamento sucessivo dos vértices adjacentes. Caso a célula adjacente desejada esteja ocupada, um laço interno é acionado para identificar a primeira célula disponível na vizinhança de C_a, respeitando as restrições de ocupação da matriz. Após o posicionamento, o vértice b passa a atuar como referência para a iteração subsequente.

A Figura 2.5 (b) apresenta uma variação da implementação, na qual o grafo de dados é representado como uma lista de arestas. Nesse modelo, as n primeiras arestas asseguram a inclusão inicial dos n vértices do grafo, o que promove uma travessia

mais sistemática e controlada. Em cada iteração, uma única aresta é processada, e o vértice de destino é posicionado de forma adjacente ao vértice de origem, observadas as restrições impostas pela ocupação da matriz.

Essa abordagem facilita a implementação. Adicionalmente, a ordenação das arestas fornecidas ao algoritmo pode ser definida por estratégias de busca em grafos, como busca em profundidade (DFS), busca em largura (BFS) e o algoritmo *zigzag*, proposto em [14], as quais influenciam diretamente a qualidade do posicionamento obtido.

Desafios para a implementação do YOTO em *hardware*

O algoritmo YOTO realiza escolhas aleatórias em diferentes etapas de sua execução, como a seleção do vértice inicial e a escolha entre células adjacentes disponíveis. Essas decisões requerem a utilização de geradores pseudoaleatórios de boa qualidade estatística. Em implementações em *hardware*, essa funcionalidade é normalmente provida por registradores de deslocamento com realimentação linear (LFSRs). Como alternativa, pode-se fornecer ao circuito uma sequência previamente aleatória de células a serem visitadas, de modo que o acesso ocorra de forma determinística e elimine a necessidade de gerar números aleatórios em tempo de execução.

Em cada iteração do algoritmo, torna-se necessário identificar quais células vizinhas à posição atual de um determinado nó estão disponíveis para alocação. Essa verificação é restrita à vizinhança da célula atualmente ocupada pelo nó. Uma estratégia simples para esse procedimento consiste na geração de deslocamentos relativos $(\Delta x, \Delta y)$, aplicados às coordenadas atuais (x, y) do nó, organizados de forma incremental em função da distância, começando pelas células a uma distância igual a 1 e, posteriormente, para distâncias maiores, conforme necessário.

A estrutura de mapeamento C_2N permite verificar se uma célula específica está ocupada ou livre e viabiliza a validação das posições candidatas durante o processo de posicionamento.

A versão do algoritmo apresentada na Figura 2.5 (b) simplifica a implementação ao operar diretamente sobre uma lista de arestas que contém n elementos, na qual todos os vértices do grafo estão presentes. Essa organização reduz a complexidade da implementação em *hardware*.

2.3.2 O Algoritmo de Travessia Dupla (YOTT)

O algoritmo YOTT [15] constitui uma extensão do algoritmo YOTO, concebida com o objetivo de assegurar a cobertura completa das arestas durante a travessia do grafo. No YOTO, cada vértice é visitado apenas uma única vez ao longo do processo, o que implica que, em grafos nos quais o número de arestas excede o número de vértices, parte das conexões não é explorada. Nesse algoritmo, o posicionamento de um vértice

ocorre no momento de sua primeira visita e baseia-se apenas na informação do vértice adjacente previamente visitado.

Essa estratégia apresenta uma natureza gulosa e baixa complexidade computacional; entretanto, pode resultar em alocações não ótimas para vértices associados a mais arestas, uma vez que informações relevantes sobre outras conexões do grafo não são consideradas no momento do posicionamento. Em contraste, o YOTT introduz mecanismos de anotação nas arestas que permitem a incorporação de informações adicionais de localidade ao longo da travessia, o que reduz a dependência de decisões aleatórias e favorece posicionamentos mais consistentes com a topologia global do grafo.

A Figura 2.6 (a) ilustra um exemplo de posicionamento prejudicado pela ausência de cobertura completa das arestas, característica do algoritmo YOTO. Nesse caso, a aresta não visitada $6 \rightarrow 1$, representada por uma linha tracejada, resulta na formação de um caminho excessivamente longo. Em contraste, a Figura 2.6 (b) apresenta a abordagem adotada pelo algoritmo YOTT, na qual anotações são associadas às arestas ao longo do processo de travessia.

No YOTT, todas as arestas do grafo são visitadas em uma primeira etapa, durante a qual são geradas anotações sempre que um vértice é alcançado por mais de uma aresta. Essas anotações introduzem restrições adicionais de localidade, com o objetivo de orientar decisões futuras de posicionamento. Na etapa seguinte, o algoritmo executa uma segunda travessia com comportamento análogo ao do YOTO; contudo, as decisões de alocação passam a considerar as anotações previamente registradas, o que permite posicionar vértices de modo a preservar a proximidade em relação a mais de um vértice.

Como exemplo, a aresta $5 \rightarrow 6$ recebe a anotação *dist 1 v1*, a qual indica que o vértice 6 deve ser posicionado a uma distância igual a 1 em relação ao vértice 1. De forma análoga, a aresta $4 \rightarrow 1$ é anotada com *dist 2 v1*, o que sinaliza a necessidade de manter uma distância igual a 2 em relação ao mesmo vértice. Essas anotações permitem ao algoritmo incorporar informações provenientes de outras arestas, coletadas durante a primeira travessia completa do grafo, ao processo de posicionamento incremental realizado na segunda travessia.

Portanto, durante a travessia do grafo da matriz, o algoritmo YOTT utiliza as anotações previamente associadas às arestas para orientar o processo de posicionamento dos vértices. A Figura 2.6 (c) ilustra uma etapa intermediária da execução do algoritmo, na qual o vértice 5 apresenta duas posições candidatas para alocação, indicadas em cinza. Entretanto, apenas uma dessas posições satisfaz a restrição imposta pela anotação correspondente, a qual estabelece que o vértice 5 deve manter uma distância igual a 2 em relação ao vértice 1. Dessa forma, a decisão de posicionamento é guiada pelas informações topológicas acumuladas durante a primeira travessia de anotação do

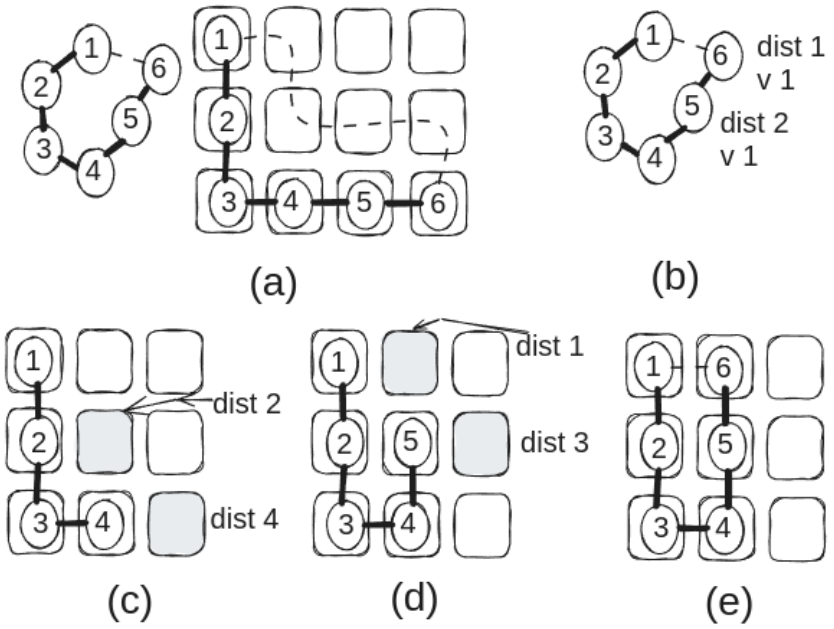


Figura 2.6: (a) Grafo de Exemplo, YOTO e posicionamento ruim; (b) Grafos com anotações; (c) Posicionamento da aresta $4 \rightarrow 5$; (d) Posicionamento da aresta $5 \rightarrow 6$; (e) Posicionamento ótimo final.

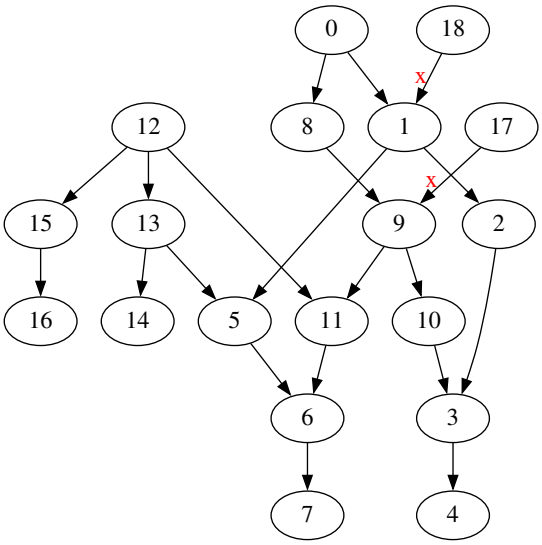
grafo.

De maneira análoga, a Figura 2.6 (d) apresenta o posicionamento do vértice 6, o qual também possui duas posições viáveis na matriz. Novamente, apenas uma dessas opções atende à restrição definida pela anotação previamente associada, que garante a relação de proximidade implícita no grafo original seja preservada no mapeamento físico.

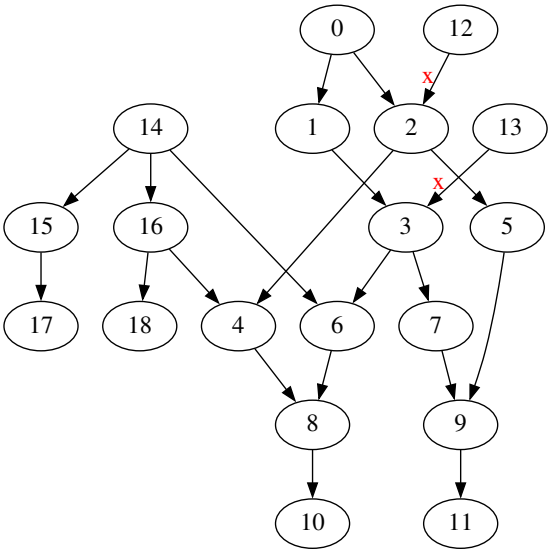
Nesse exemplo específico, o posicionamento final corresponde a uma solução ótima. As decisões de alocação foram orientadas pelas anotações, as quais foram geradas durante a primeira travessia completa do grafo. O algoritmo mantém complexidade polinomial linear, passando de $O(n)$ para $O(n + a)$, em que n representa o número de vértices e a o número de arestas do grafo, em razão da necessidade de visitar todas as arestas ao longo das travessias..

2.4 Travessia *ZigZag*

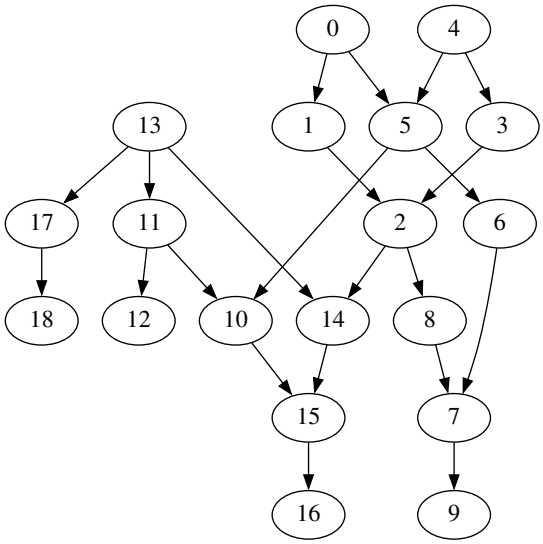
A travessia do grafo por meio do percurso em *zigzag* [16] tem como objetivo identificar e explorar regiões reconvergentes. O princípio do método baseia-se em um percurso em profundidade de modo alternado nos sentidos entrada→saída e saída→entrada. Ao identificar um vértice cujo grau de saída (*fan-out*) ou de entrada (*fan-in*) seja maior que um, a direção da travessia é invertida. Essa alternância permite a identificação das reconvergências no grafo.



(a) Travessia em Profundidade DFS.



(b) Travessia em Largura BFS.



(c) Travessia em *zigzag*.

Figura 2.7: Comparação entre os caminhos percorridos por DFS em (a); BFS em (b); *zigzag* em (c).

A Figura 2.7 apresenta uma comparação entre os caminhos percorridos por três estratégias de travessia de grafos a partir do vértice 0: busca em profundidade (DFS), ilustrada na Figura 2.7(a); busca em largura (BFS), apresentada na Figura 2.7(b); e o algoritmo *zigzag*, mostrado na Figura 2.7(c).

Os vértices estão numerados de acordo com a ordem de visita em cada percurso. Observa-se que a travessia em *zigzag* identifica a região reconvergente composta pelos vértices 0, 1, 2, 3 e 4, o que auxilia o posicionamento baseado em vizinhança adotado pelos algoritmos YOTO e YOTT. Além disso, diferentemente de outras estratégias de travessia, quando o grafo é conexo, todos os vértices são percorridos com um lastro de conexão, enquanto as travessias em profundidade (DFS) e em largura (BFS) podem

gerar descontinuidades nas sequências de entrada e/ou saída dos vértices.

2.5 Arquiteturas Reconfiguráveis de Grão Grosso (CGRA)

Os CGRAs constituem uma classe de arquiteturas reconfiguráveis que operam em um nível de abstração mais elevado quando comparadas aos FPGAs. Enquanto os FPGAs realizam o processamento em nível de bit, os CGRAs operam predominantemente em nível de palavra ou de instrução, característica que favorece o mapeamento direto de grafos de fluxo de dados (DFGs) a nível de instruções aritméticas, lógicas, de controle ou de acesso à memória. Esses grafos correspondem tipicamente à saída de compiladores e, em geral, apenas trechos de código contendo laços computacionalmente intensivos são selecionados para mapeamento nesse tipo de arquitetura [84].

A estrutura básica de um CGRA é composta por dois componentes principais: os elementos de processamento (PEs) e a rede de interconexão responsável pela comunicação entre esses elementos. A forma como os PEs são organizados e interligados define a arquitetura da rede reconfigurável, e a rede exerce influência direta sobre o desempenho e a flexibilidade da arquitetura.

O processo de mapeamento de um DFG em um CGRA envolve três etapas. A primeira consiste no posicionamento, no qual os vértices do DFG, como instruções, entradas e saídas, são atribuídos aos PEs da arquitetura. A segunda etapa corresponde ao roteamento, responsável por implementar as arestas com recursos de interconexão disponíveis. Por fim, o escalonamento define a temporização das operações de computação e comunicação.

Os CGRAs podem ser classificados de diversas formas; uma delas é usar o número de dimensões empregadas na organização espacial de seus elementos de processamento, sendo comuns as arquiteturas unidimensionais (1D) e bidimensionais (2D). A Figura 2.8 apresenta exemplos de topologias. Inicialmente, são ilustradas duas topologias unidimensionais: uma baseada em anel ou *pipeline* e outra baseada em barramento (*bus*), conforme mostrado na Figura 2.8 (a).

A topologia bidimensional em grade ou malha (*mesh*) é a mais difundida e ilustrada na Figura 2.8 (b). Nessa configuração, cada elemento de processamento conecta-se aos seus quatro vizinhos imediatos nas direções norte, sul, leste e oeste. A malha pode ser estendida com a inclusão de conexões de salto. A topologia 1-*hop* adiciona ligações diretas a elementos localizados a uma distância adicional de um salto em cada direção, conforme ilustrado na Figura 2.8 (c). A inclusão de quatro ligações adicionais reduz as distâncias médias de comunicação em aproximadamente um fator de dois em relação a malha. Os experimentos conduzidos nesta tese avaliaram as topologias de malha e 1-*hop*.

O principal desafio associado ao escalonamento espacial está na necessidade de

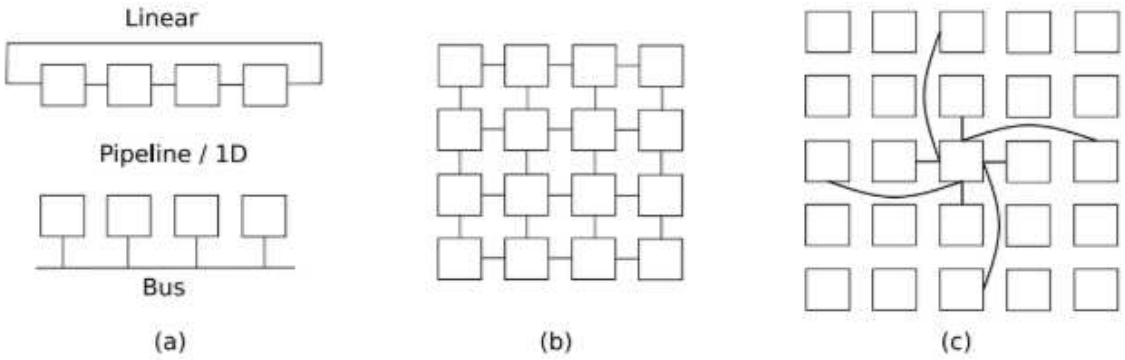


Figura 2.8: (a) Anel ou *Pipeline* e *BUS*; (b) Arquitetura *mesh*; (c) Conexão da Arquitetura 1-*hop*

balanceamento dos grafos de fluxo de dados, de modo a preservar a correta sincronização. Todos os caminhos entre vértices produtores e consumidores devem apresentar comprimentos equivalentes para assegurar que os dados cheguem de forma simultânea. Essa restrição aumenta a complexidade do mapeamento, mesmo para grafos com poucos vértices, tipicamente 30 ou menos [84].

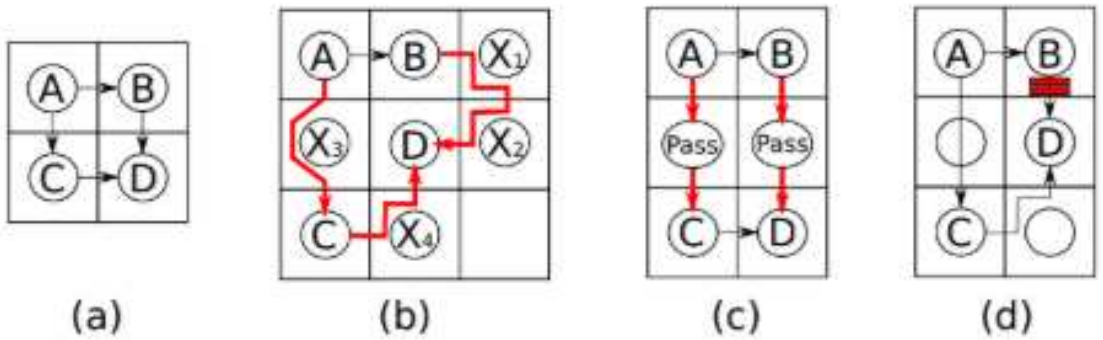


Figura 2.9: (a) Solução Ótima; (b) Dois caminhos balanceados; (c) Outro exemplo de dois caminhos balanceados; (d) Uso de fifos ou buffers para balancear o caminho mais curto.

A Figura 2.9 ilustra quatro situações distintas de posicionamento, roteamento e escalonamento para um grafo simples em uma arquitetura em malha. A Figura 2.9 (a) apresenta um posicionamento ótimo, no qual as etapas de roteamento e escalonamento são automaticamente satisfeitas. Nesta tese, busca-se a implementação de soluções de posicionamento ótimas ou tão próximas quanto possível do ótimo, por meio do emprego de heurísticas.

A Figura 2.9 (b) apresenta um exemplo no qual são utilizados caminhos de roteamento mais longos. No percurso que parte de a e passa por c, o sinal atravessa uma célula intermediária, alcança c, passa por uma segunda célula intermediária e, por fim, chega a d. No percurso alternativo, ao sair de a, o sinal alcança b, atravessa duas

células intermediárias e chega igualmente a d. Em ambos os casos, o comprimento total dos caminhos $a \rightarrow d$, através de c ou por b, é igual a 3. Dessa forma, obtém-se um escalonamento válido, análogo ao apresentado na Figura 2.9 (c).

Por outro lado, na Figura 2.9 (d), o caminho $a \rightarrow b \rightarrow d$ apresenta um comprimento inferior ao percurso alternativo. Para que o escalonamento permaneça válido, torna-se necessária a inserção de elementos de balanceamento temporal, como *buffers* ou filas, destacados em vermelho na Figura 2.9 (d).

2.6 *Field-Programmable Gate Array* (FPGA)

Os *Field-Programmable Gate Arrays* (FPGAs) são circuitos integrados reconfiguráveis. Diferentemente dos *Application-Specific Integrated Circuits* (ASICs), cuja funcionalidade permanece fixa após a fabricação, os FPGAs permitem reconfiguração das conexões e das funções lógicas, o que possibilita a implementação de soluções personalizadas para cada aplicação.

Em relação às arquiteturas CGRA, destaca-se que, enquanto essas realizam o mapeamento de grafos de código, os FPGAs realizam o mapeamento de grafos de circuitos. Esses grafos podem alcançar milhares ou até centenas de milhares de nós, o que caracteriza um aumento significativo de escala, além disso, a proporção de conexões e reconvergências é maior nos grafos para FPGAs. Contudo, diferentemente dos CGRAs, não é necessário garantir o balanceamento dos caminhos. Nesse contexto, métricas de custo distintas são empregadas, com ênfase na redução do caminho crítico e na minimização do uso excessivo dos recursos de interconexão.

Cada bloco lógico em um FPGA é composto por uma ou mais *Lookup Tables* (LUTs), multiplexadores e registradores, e pode incluir blocos dedicados, como memórias B-RAMs e unidades de processamento digital de sinais (DSPs) [15].

O desenvolvimento de projetos em FPGAs requer o uso de ferramentas específicas, conhecidas como ferramentas de CAD, que englobam etapas como síntese lógica, mapeamento tecnológico, posicionamento (*placement*) e roteamento (*routing*) [64]. Entre as ferramentas de CAD disponíveis, destaca-se o projeto VTR, que oferece um fluxo completo de código aberto, com suporte ao sintetizador ODIN-II, ao mapeador lógico ABC e ao posicionador e roteador VPR [64].

Ao longo das últimas décadas, os FPGAs passaram por uma evolução significativa com a incorporação de arquiteturas heterogêneas, com milhões de LUTs, milhares de DSPs e milhares de blocos B-RAM. Nesta tese, o foco recai sobre o mapeamento em LUTs. Dessa forma, são avaliados circuitos combinacionais de diferentes tamanhos, desde dezenas até dezenas de milhares de portas lógicas, com o objetivo de analisar a escalabilidade dos algoritmos de posicionamento em *hardware* e o impacto das estruturas de dados associadas.

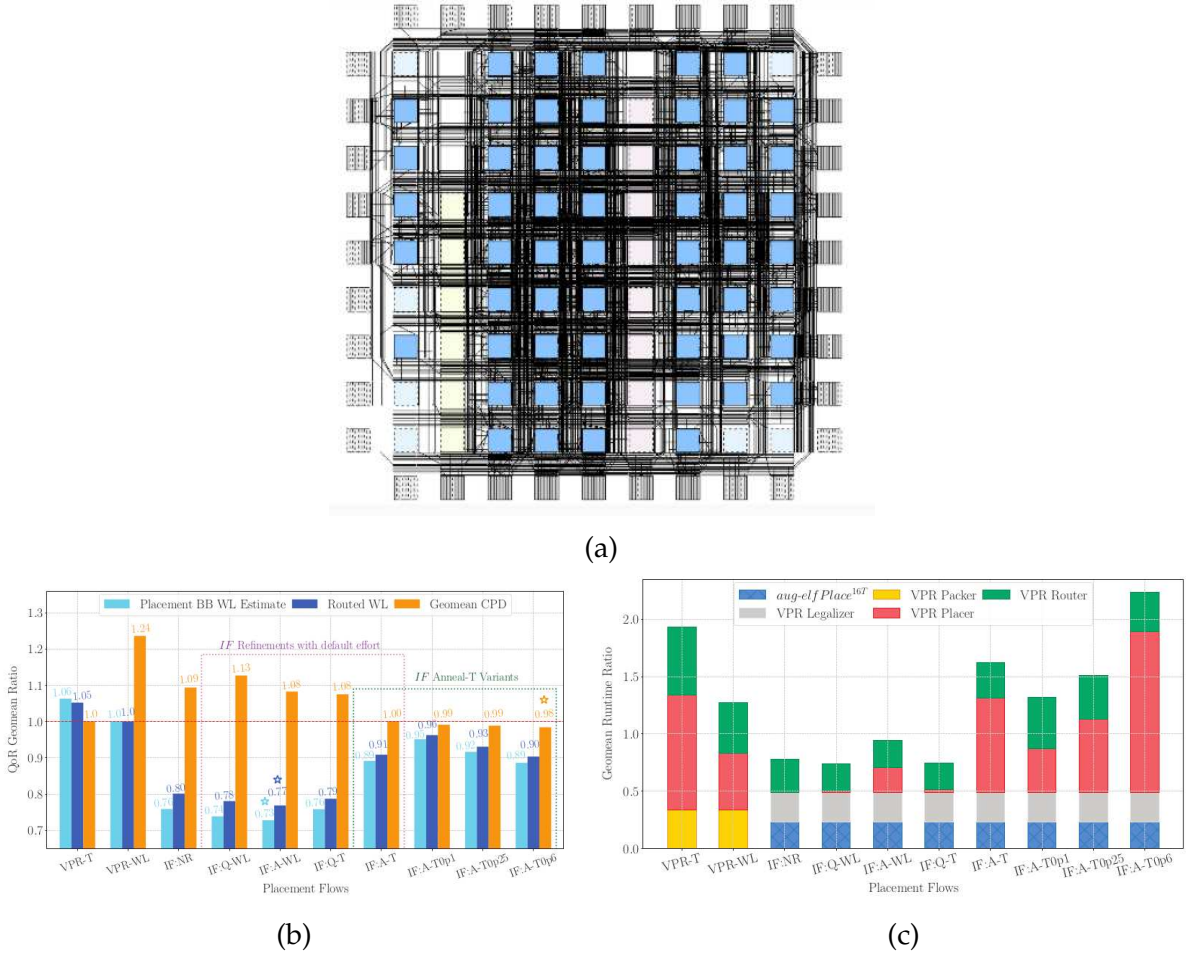
Recentemente [38], observou-se um interesse crescente no uso exclusivo de LUTs de FPGAs para a inferência de redes neurais profundas (DNNs), com o objetivo de obter baixa latência e elevada eficiência energética. Essa abordagem é particularmente relevante para aplicações de borda, nas quais soluções baseadas em computação em nuvem apresentam limitações associadas à latência e à segurança. Em contrapartida, implementações tradicionais de DNNs em FPGAs dependem fortemente de blocos DSP para operações de multiplicação e acumulação (MAC), o que restringe a escalabilidade. As abordagens baseadas exclusivamente em LUTs [38] buscam contornar essa limitação por meio do aproveitamento integral desses recursos para uma melhor utilização da arquitetura e redução da latência de inferência. Assim, o problema de mapeamento eficiente em LUTs apresenta relevância na computação de borda.

Nesta tese, os resultados obtidos são comparados com aqueles produzidos pelo VPR [11], uma ferramenta que tem sido continuamente aprimorada [69]. Entre os trabalhos relacionados, destaca-se o HCAS [42], que emprega um estágio de posicionamento global analítico, seguido de legalização e refinamento com SA e atinge reduções de 3% no comprimento dos fios e 5% no caminho crítico, com tempo de posicionamento até 10 vezes inferior ao do VPR7. Entretanto, os experimentos do HCAS restringiram-se a projetos de pequeno porte, com menos de 31 mil blocos lógicos. Em comparação com o VPR7+, o VPR8 obteve reduções de 33% no comprimento dos fios e de 11% no caminho crítico [64]. O algoritmo StarPlace, por sua vez, utiliza um modelo analítico quase-linear baseado em uma rede do tipo *star+* que permite a redução de 9% no caminho crítico e um tempo de posicionamento cinco vezes inferior ao do VPR nos *benchmarks* MCNC. Sua versão acelerada por GPU atingiu reduções de 26% no caminho crítico, com tempo de posicionamento até 77 vezes inferior ao de uma versão do VPR7 orientada pelo comprimento dos fios.

Dado que o VPR oferece um amplo conjunto de parâmetros de otimização, seu tempo de execução pode variar significativamente, desde alguns segundos até várias horas para um mesmo circuito. Nesta tese, opta-se por selecionar configurações do VPR que reduzam ao máximo o tempo de execução, porém com a obtenção de soluções válidas para fins de comparação.

A Figura 2.10 mostra que o VTR permite a otimização do fluxo de projeto, com foco no tempo ou no comprimento dos fios. Observa-se que o atraso do caminho crítico pode ser significativamente reduzido; entretanto, essa otimização resulta em um aumento substancial no tempo total de posicionamento. Mesmo abordagens recentes apresentam dificuldades em reduzir simultaneamente o tempo de posicionamento e o atraso do caminho crítico [69].

Nota-se que o tempo de posicionamento, indicado em vermelho na figura, domina o tempo total de execução, independentemente da estratégia empregada, seja o SA ou métodos de posicionamento analítico, representados pelas cores cinza e azul. Em



(a)

(b)

(c)

Figura 2.10: (a) Exemplo de FPGA com roteamento, onde observa-se a alta densidade dos grafos de conexões, sendo mais importante minimizar o caminho crítico ou uso de fios do que o balanceamento dos caminhos; (b) Comparação de desempenho com a barra laranja que indica o caminho crítico para VTR-T (otimizado por tempo) e VTR-WL (otimizado por fio) que indica que o VTR-T é 25% melhor que VTR-WL, enquanto as variações recentes [69] são piores ou no máximo equivalentes ao VTR-T; Figura Extraída [69]; (c) Tempo de execução normalizado, com posicionamento em vermelho/cinza/azul e roteamento em verde que mostra que o posicionamento é o processo dominante e é o foco deste trabalho. Figura Extraída [69]

contraste, o tempo de roteamento, destacado em verde, apresenta uma contribuição relativamente menor para o tempo total do fluxo.

2.7 *Quantum-dot Cellular Automata* (QCA)

Os *Quantum-dot Cellular Automata* (QCAs) são uma nanotecnologia proposta na década de 1990 com o objetivo de superar os limites físicos associados à miniaturização contínua das tecnologias baseadas em transistores CMOS [49]. Diferentemente dos circuitos convencionais, os QCAs não utilizam corrente elétrica para a transmissão de informação. Em vez disso, os estados lógicos são representados pela configuração espacial de elétrons em estruturas quânticas denominadas células QCA, nas quais o valor lógico é determinado pelo arranjo de repulsão eletrostática entre cargas confinadas em pontos quânticos [80].

O modelo computacional baseado em QCA impõe desafios específicos ao processo de mapeamento de circuitos lógicos, ao posicionamento físico dos elementos e à sincronização temporal por meio de esquemas de *clocking*. Em função do mecanismo de propagação de sinais, fundamentado em interações eletrostáticas entre células adjacentes, a disposição espacial adequada dos elementos assume um papel central para o funcionamento correto dos circuitos.

Além disso, as implementações físicas em QCA exigem o balanceamento dos caminhos de propagação. Dessa forma, embora o mapeamento para QCA trate de circuitos lógicos, o problema se assemelha mais ao mapeamento para arquiteturas CGRA.

2.7.1 Estrutura Básica de Circuitos QCA

No QCA, a porta inversora (*NOT*) pode ser implementada com células dispostas diagonalmente, onde a célula subsequente assume uma polarização oposta à da célula precedente como ilustrada na Figura 2.11 [80].

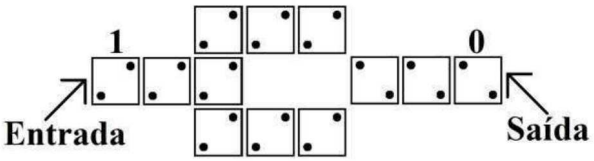


Figura 2.11: Estrutura da porta inversora com a utilização de células QCA [80].

Diferente do CMOS que tipicamente utiliza a porta *NAND* como base, no QCA é a porta maioria (*MAJ*). A função lógica realizada por essa estrutura é definida por $MAJ(A, B, C) = AB + AC + BC$. Por meio da fixação de uma das entradas em 0 ou 1,

derivam-se, respectivamente, as funções lógicas *AND* e *OR*, as quais, em conjunto com a porta *NOT*, são capazes de formar o conjunto funcional lógico da tecnologia QCA. A estrutura clássica da porta maioria foi proposta em [80] e encontra-se ilustrada na Figura 2.12.

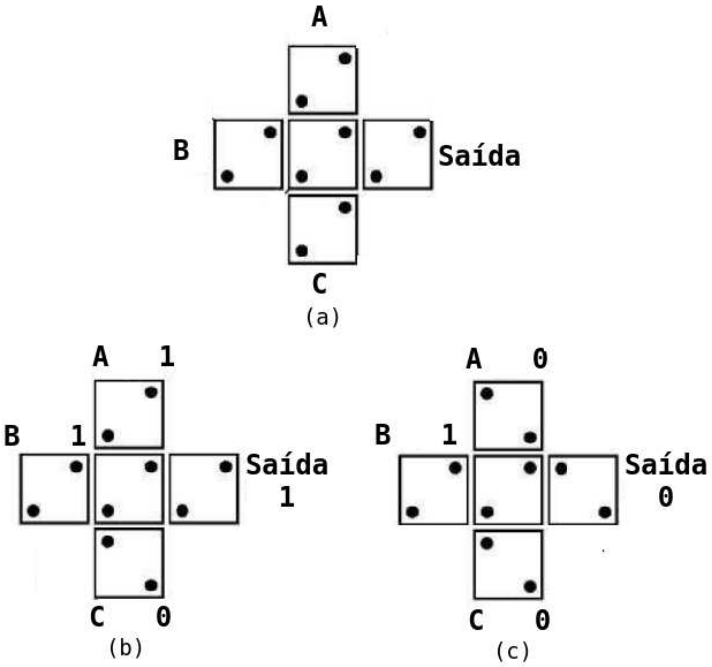


Figura 2.12: Estrutura da porta maioria (*MAJ*) em tecnologia QCA [80].

2.7.2 O Esquema de *Clock* USE

Diferentemente dos circuitos baseados em CMOS, a tecnologia QCA requer um sistema de sincronismo, mesmo para circuitos combinacionais. Diversos esquemas de sincronismo foram propostos na literatura; neste trabalho, optou-se por avaliar os sistemas USE e *2DDWave*. O problema de posicionamento e roteamento em QCA é classificado como NP-difícil [89].

O sistema USE [12] permite a propagação de sinais em duas dimensões por meio de uma estrutura escalável de numeração das fases do *clock*. Nesse modelo, o sinal percorre o circuito e obedece a uma sequência cíclica crescente de fases, no sentido $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1 \rightarrow \ldots$, conforme ilustrado na Figura 2.13. De forma resumida, o USE possui uma alternância no sentido das conexões: nas linhas, a propagação ocorre de modo alternado da esquerda para a direita e vice-versa, enquanto, nas colunas, o mesmo ocorre na vertical.

Para ilustrar o mapeamento de um circuito simples, considera-se inicialmente um meio-somador, cujo grafo estrutural é apresentado na Figura 2.14. Esse grafo é, então, mapeado de acordo com as restrições impostas pelo esquema de *clock* USE e pelo balanceamento, conforme indicado pelas conexões pontilhadas.

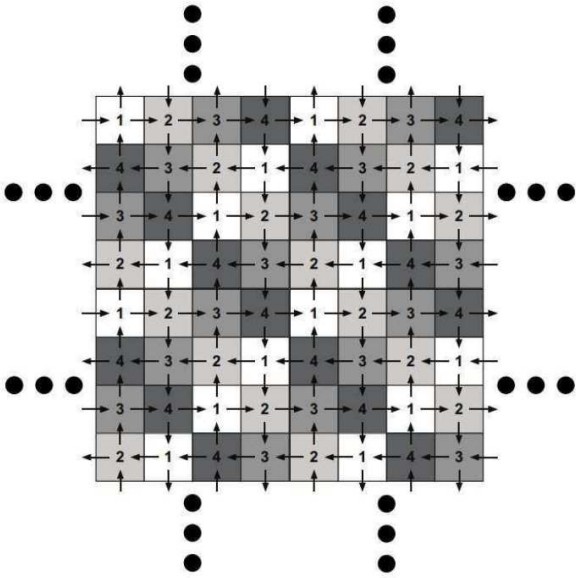


Figura 2.13: Estrutura da arquitetura USE [12]

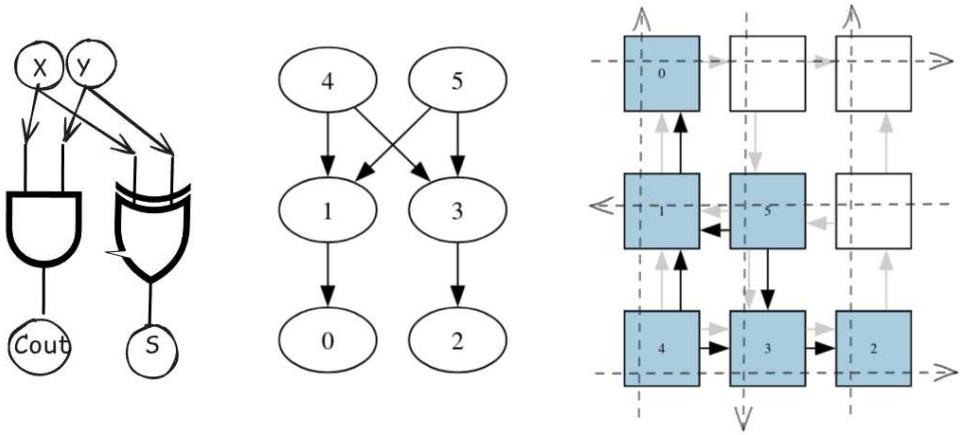


Figura 2.14: Exemplo de um meio somador ou *half adder* no esquema de *clock* USE

2.7.3 O Esquema de *Clock 2DDWave*

O esquema de *clock 2DDWave* utiliza duas direções principais de propagação dos sinais: uma no sentido horizontal, da esquerda para a direita, e outra no sentido vertical, de cima para baixo.

Na Figura 2.15, apresenta-se o mesmo exemplo do meio-somador com o esquema de *clock 2DDWave*. Inicialmente, o grafo do circuito é modificado de modo a incluir vértices de *fan-out* que possuem, na Figura, uma entrada e duas saídas, vértices de cruzamento (representados por "+") e vértices de *buffer* (destacados em amarelo). O principal desafio consiste em restringir a propagação dos sinais exclusivamente às direções permitidas pelo esquema *2DDWave*.

Embora a Figura 2.15 apresente um posicionamento e roteamento válidos para o meio-somador, o circuito resultante não está balanceado. Em particular, o vértice

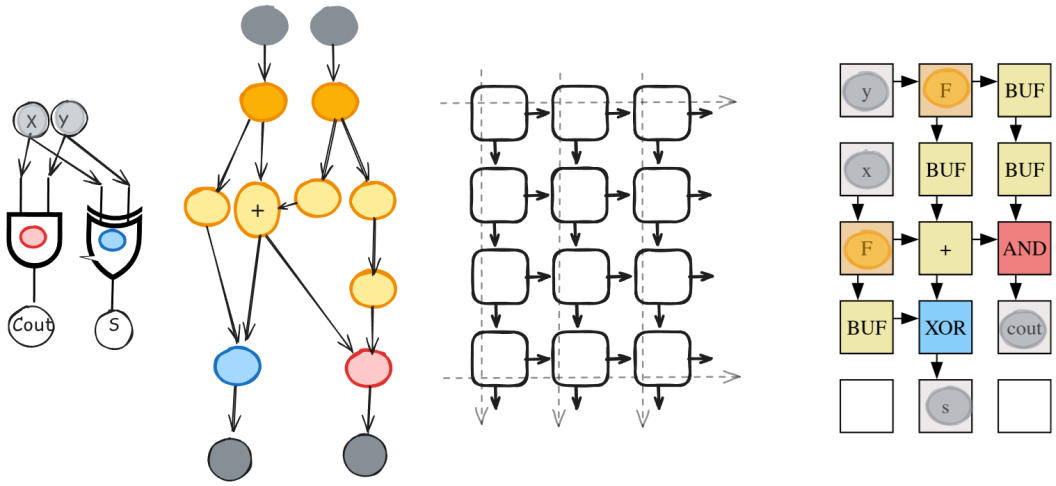


Figura 2.15: Estrutura da arquitetura *2DDWave* [12] com o exemplo do meio somador

de entrada x percorre caminhos de comprimentos distintos em relação ao vértice de entrada y até as saídas s e c_{out}, o que viola a exigência de sincronismo entre os sinais.

Capítulo 3

Aceleradores em Hardware

3.1 Posicionamento em Diferentes Contextos

Conforme discutido na Seção 2.1, o problema de posicionamento é classificado como *NP-difícil*, independente das restrições e métricas de custo que iremos abordar, que variam de acordo com a arquitetura alvo. Nesta tese, o problema foi testado em três áreas distintas: CGRAs, FPGAs e QCA. As subseções a seguir descrevem as métricas em cada contexto.

3.1.1 Posicionamento em CGRA

Em arquiteturas *Coarse-Grained Reconfigurable Arrays* CGRAs, além do posicionamento espacial, o processo de mapeamento envolve o escalonamento temporal das operações, com o objetivo de balancear os caminhos do grafo e garantir a sincronização do fluxo de dados ao longo da execução, onde podem ser empregadas filas de balanceamento. A imposição de limites ao número máximo de filas pode reduzir o custo, mas aumenta a complexidade do processo de mapeamento [65].

Diversas abordagens foram propostas na literatura para tratar esses desafios. Métodos baseados em Programação Inteira Linear (ILP) [84] e resolvedores SAT [27] produzem soluções exatas; porém, apresentam limitações de escalabilidade e podem ser aplicados, geralmente, apenas a grafos com até aproximadamente 30 vértices. Outras estratégias exploram técnicas de *modulo scheduling* aplicadas a arquiteturas com número reduzido de PEs [30], o que impõe restrições de desempenho devido à multiplexação temporal das tarefas.

Abordagens híbridas também foram investigadas [65] com a combinação de heurísticas e meta-heurísticas, como ILP, algoritmos gulosos, algoritmos genéticos [25], aprendizado por reforço [54] e redes neurais aplicadas a grafos (GNN) [53]. Apesar dos avanços alcançados, essas técnicas ainda enfrentam dificuldades de escalabilidade e demandam tempos de execução elevados.

Entre as estratégias mais eficazes, destaca-se o uso do algoritmo *Simulated Annealing* SA como etapa de posicionamento [93, 17, 64]. Entretanto, esse algoritmo exige um

número expressivo de iterações, o que impacta o tempo de execução. Em contraposição, algoritmos baseados em travessia de grafos apresentam estruturas mais simples e tempos de execução significativamente reduzidos.

Nesta tese, são adaptadas para o domínio dos CGRAs as versões em *hardware* dos algoritmos SA, YOTO e YOTT. Os experimentos e resultados obtidos podem ser vistos no Capítulo 4 Seção 4.1.

3.1.2 Posicionamento em FPGA

Nesta tese, nosso foco para os FPGAs foi colocado sobre o posicionamento de blocos lógicos LUTs. Trabalhos recentes voltados à computação de borda têm explorado a geração de circuitos baseados exclusivamente em LUTs [38]. Diferentemente das arquiteturas CGRAs, nas quais a ênfase recai sobre a sincronização e o balanceamento da vazão, os algoritmos de posicionamento para FPGAs devem lidar com circuitos digitais compostos por milhares ou dezenas de milhares de elementos interconectados.

As versões em *hardware* dos algoritmos SA, YOTO e YOTT, inicialmente concebidos para arquiteturas CGRAs, foram adaptadas para o domínio dos FPGAs. Como principais contribuições, destacam-se a reestruturação dos *pipelines* de execução, a reorganização das estruturas de memória e a introdução de mecanismos de *cache*, com o objetivo de reduzir o consumo de B-RAMs e viabilizar a implementação paralela em dispositivos com recursos limitados. Os experimentos e resultados obtidos podem ser vistos no Capítulo 4 Seção 4.2

3.1.3 Posicionamento em QCA

Um dos principais desafios no mapeamento de circuitos em QCA está associado ao balanceamento dos caminhos. Estas restrições tornam o problema conceitualmente semelhante ao posicionamento em arquiteturas CGRAs, entretanto os recursos de roteamento são mais restritos. Além disso, os esquemas de *clocking* influenciam diretamente o uso de mais recursos (fios) para o roteamento, que representam uma parcela significativa da área total dos circuitos [78].

Diversas heurísticas foram propostas para mitigar essas limitações [78]. Métodos exatos baseados em SMT Solvers também foram explorados, o que permitiu a avaliação simultânea de múltiplos objetivos e restrições físicas [88]. Contudo, tais abordagens apresentam limitações de escalabilidade.

Nesta tese, propõe-se a adaptação e a avaliação dos algoritmos YOTO e YOTT em arquiteturas QCA, além de avaliar uma versão do SA, que já foi usado em QCA [50]. Em particular, A versão proposta introduz uma nova função de custo para o algoritmo SA que incorpora penalidades específicas associadas às restrições topológicas e de

adjacência impostas pelas células QCA. Os experimentos e resultados obtidos podem ser vistos no Capítulo 4 Seção 4.3.

3.2 Acelerador em Hardware para *Simulated Annealing*

O algoritmo *Simulated Annealing* (SA) apresenta características intrinsecamente sequenciais conforme visto na Seção 2.2. Para contorná-las, esta tese adota uma abordagem baseada na exploração combinada de paralelismo temporal e espacial, por meio da organização do algoritmo em uma arquitetura *pipeline* com múltiplas *threads* independentes. Essa estratégia foi inicialmente desenvolvida e validada para arquiteturas CGRAs e, posteriormente, adaptada e estendida para o domínio de FPGAs.

A principal ideia consiste em decompor o fluxo do algoritmo SA em estágios *pipeline*. Embora os estágios sejam sequenciais, a execução é sobreposta no tempo, que permite a execução de diferentes instâncias do algoritmo, representadas por *threads* distintas, explorando o paralelismo temporal.

Além disso, sempre que possível, operações independentes dentro de um mesmo estágio são executadas em paralelo, de modo a explorar o paralelismo espacial por meio da replicação de unidades funcionais e do acesso concorrente às estruturas de memória.

3.2.1 Pipeline do SA para CGRAs

A arquitetura proposta executa múltiplas instâncias independentes do algoritmo SA, cada uma associada a uma *thread*. Cada *thread* mantém seu próprio estado de execução, inclusive as estruturas privadas de mapeamento entre nós e células. Essa independência permite que diferentes regiões do espaço de soluções sejam exploradas simultaneamente, de modo a aumentar a diversidade das soluções candidatas, característica já explorada em trabalhos anteriores como [17].

No que se refere às estruturas de dados utilizadas, destaca-se que apenas as tabelas de mapeamento N_2C (nó para célula) e C_2N (célula para nó) necessitam de atualização durante a efetivação de uma troca entre dois nós a e b. Tal característica permite uma implementação mais simples.

A Figura 3.1 apresenta uma visão geral do fluxo de execução da versão em *pipeline* do algoritmo SA. Inicialmente, duas células da malha de posicionamento são selecionadas de forma pseudoaleatória. No estágio subsequente, a estrutura de mapeamento C_2N é acessada para identificar os nós atualmente alocados nessas células. A sequência de estágios seguintes implementa diretamente, em *hardware*, as operações descritas no pseudocódigo da Figura 2.3 (a) e de acordo com a ordem lógica do algoritmo.

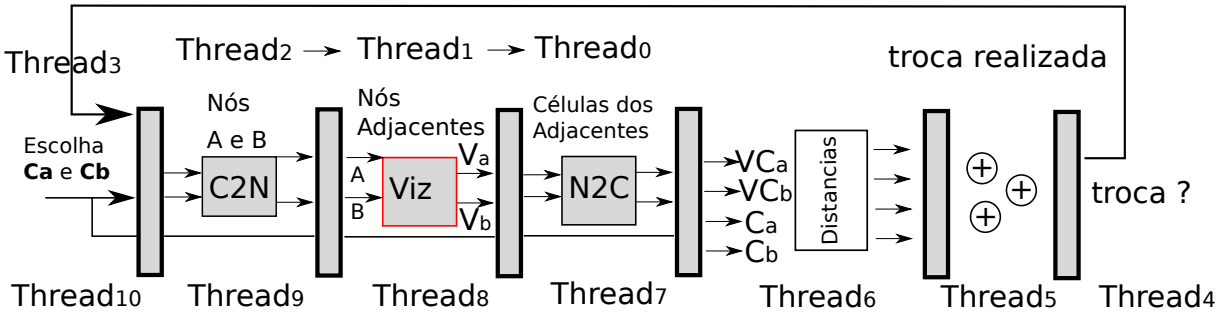


Figura 3.1: *Pipeline* simplificado com múltiplas *threads*.

O paralelismo temporal é alcançado por meio da execução simultânea de várias *threads*, cada uma associada a uma instância independente do algoritmo. Dessa forma, enquanto uma *thread* executa a etapa de avaliação e decisão quanto à aceitação de uma troca, outras realizam operações distintas, como o cálculo de distâncias, o acesso às estruturas de memória ou a atualização parcial das estruturas de posicionamento.

Cada estágio do *pipeline* corresponde a uma operação específica do algoritmo, e cada acesso às estruturas de memória é assumido como realizado em um único ciclo de *clock*. Para atender às restrições de escrita impostas pelas memórias internas do FPGA, em particular a limitação de uma operação de escrita por ciclo, as *threads* são inseridas no *pipeline* com espaçamento de dois ciclos. Essa estratégia permite que as atualizações das estruturas de mapeamento sejam realizadas de forma consistente, sem violar as restrições de acesso à memória, e garante a correta sobreposição temporal das execuções das *threads* ao longo do *pipeline*.

Uma vantagem adicional dessa organização reside na distribuição das estruturas de dados ao longo do *pipeline*, o que contribui para a paralelização das memórias do FPGA. As subseções a seguir detalham a organização dos estágios do *pipeline*.

3.2.2 Memórias

A maior parte dos dados utilizados na execução do algoritmo é armazenada em memórias internas do FPGA, distribuídas de forma a acompanhar os diferentes estágios do *pipeline* e possibilitar o acesso paralelo. Essas memórias podem ser implementadas tanto por meio de blocos dedicados de memória (B-RAMs) quanto a partir da lógica programável do dispositivo, como LUTs. A utilização de B-RAMs apresenta vantagens, como o acesso paralelo e o aproveitamento de recursos nativos do dispositivo. Essa característica permite a redução da lógica de controle associada e diminui a necessidade de multiplexadores adicionais.

Além disso, muitos FPGAs modernos oferecem suporte à configuração de blocos de memória com múltiplas portas de leitura e uma porta de escrita por ciclo de *clock*, o que viabiliza o acesso concorrente aos dados por diferentes estágios do *pipeline*.

3.2.3 Armazenamento do Grafo

Adotaremos a seguinte notação para nó/vértice e célula. O termo nó refere-se aos vértices do DFG, enquanto célula corresponde às posições físicas disponíveis na matriz da arquitetura-alvo, nas quais os nós são alocados.

O grafo (DFG) é armazenado na forma de lista de adjacência. Nesta representação, para cada nó do grafo, mantém-se uma lista que contém os identificadores de seus nós adjacentes. Como o grafo permanece inalterado ao longo da execução do algoritmo, a memória responsável pelo armazenamento das listas de adjacência pode ser compartilhada entre todas as *threads*, sem risco de conflitos de leitura. Na arquitetura proposta, essa estrutura corresponde à memória denominada "Viz" (vizinhos/adjacentes), ilustrada na Figura 3.1.

Nos grafos avaliados nesta pesquisa, os operadores representados pelos nós são predominantemente unários ou binários, o que resulta em um grau de entrada limitado a dois. Para operadores com um maior número de saídas, foram aplicadas transformações estruturais no grafo, como a inserção de nós do tipo *buffer*. Essas transformações permitem a decomposição do grafo em subestruturas com grau máximo de saída igual a dois; portanto, cada nó terá, no máximo, quatro nós adjacentes. Ainda assim, o código foi projetado de forma parametrizável, o que possibilita sua adaptação para grafos com maior *fan-in* e *fan-out*.

3.2.4 Quantidade de Módulos de Memória

Considerando o cenário no qual cada iteração do algoritmo avalia dois nós e, ao assumir que cada nó pode ter até quatro nós adjacentes, tornam-se necessárias até oito leituras simultâneas para o processamento completo das conexões envolvidas. Entretanto, a maioria dos módulos de memória disponíveis em FPGAs oferece, no máximo, duas portas de leitura por ciclo de *clock*, o que impõe uma limitação prática ao acesso paralelo a esses dados.

Diante dessa restrição, foram consideradas duas estratégias de implementação. A primeira baseia-se na replicação da memória que armazena as listas de adjacência ao longo de múltiplos estágios do *pipeline* para permitir a leitura de dois nós adjacentes por estágio. Dessa forma, com a utilização de quatro estágios consecutivos, torna-se possível realizar as oito leituras necessárias. A segunda estratégia consiste na duplicação da memória em um único estágio do *pipeline*, com a instância de quatro módulos independentes, cada um capaz de atender a duas leituras simultâneas. Ambas as abordagens foram implementadas no gerador de código desenvolvido nesta tese para que a configuração seja ajustada conforme as características e restrições da arquitetura-alvo. Dessa forma, para os experimentos realizados, adotou-se a segunda estratégia, uma vez que ambas apresentam o mesmo custo em termos de memória, porém a

replicação das memórias em um mesmo estágio resulta em menor profundidade do *pipeline*. Essa redução permite diminuir o número de *threads* necessárias para manter o *pipeline* preenchido.

3.2.5 Dados privados de cada *Thread*

A implementação paralela do algoritmo SA exige que cada *thread* mantenha um conjunto isolado de dados, de modo a assegurar a consistência das operações e evitar condições de corrida. Para atender a esse requisito, as memórias responsáveis pelo armazenamento das relações entre nós e células (N_2C e C_2N) foram particionadas logicamente para garantir que cada *thread* disponha de uma região exclusiva para leitura e escrita.

Esse particionamento é realizado por meio da concatenação do identificador da *thread* com o endereço lógico da célula ou do nó, conforme ilustrado na Figura 3.2. Como exemplo, as *threads* 0, 1, 2 e 3 acessam, respectivamente, as posições 2, 0, 99 e 1 de suas regiões reservadas. Essa estratégia assegura o isolamento entre as regiões de memória associadas a cada *thread* e facilita a replicação das estruturas ao longo dos estágios do *pipeline*, o que possibilita acessos simultâneos sem interferência entre instâncias paralelas.

Adicionalmente, a duplicação e a distribuição dessas memórias em diferentes estágios do *pipeline*, conforme discutido na Seção 3.2.4, permitem atender aos requisitos de paralelismo e desempenho. Essa organização viabiliza múltiplos acessos concorrentes preservando o mapeamento dos identificadores dos nós e células.

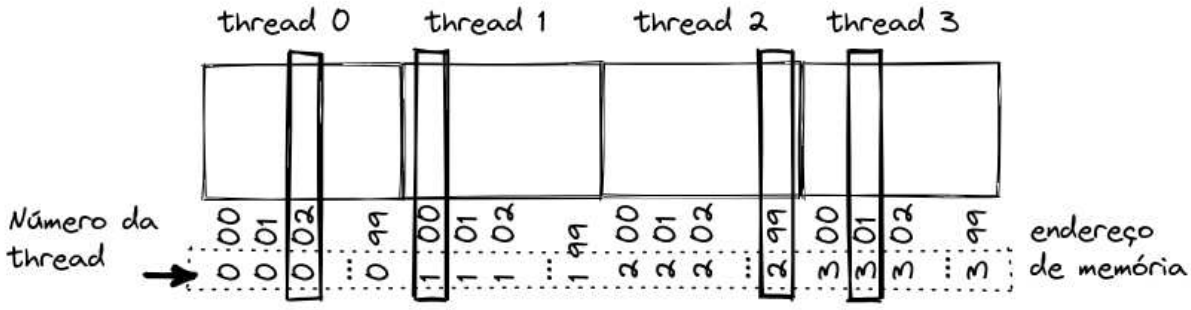


Figura 3.2: Uso das memórias com múltiplas *threads*.

3.2.6 Atualização em Dois Ciclos

Um desafio identificado durante a implementação do algoritmo SA em *hardware* está relacionado ao processo de atualização das estruturas de mapeamento N_2C e C_2N. Quando uma troca entre dois nós a e b é aceita, torna-se necessária a atualização de suas posições na matriz de posicionamento: o nó a passa a ocupar a célula C_b, enquanto o

nó b passa a ocupar a célula C_a. Dessa forma, são requeridas duas operações de escrita em cada uma das estruturas de mapeamento, a saber: $N_2C(b) \leftarrow C_a$ e $N_2C(a) \leftarrow C_b$, bem como as atualizações correspondentes na estrutura C_2N.

Entretanto, os blocos de memória B-RAMs disponíveis nos FPGAs permitem apenas uma operação de escrita por ciclo de *clock*. Para contornar essa limitação, adotou-se uma estratégia de espaçamento temporal entre as inserções das *threads* no *pipeline*. Em vez de inserir uma nova *thread* a cada ciclo, as *threads* são injetadas no *pipeline* a cada dois ciclos, o que permite que o processo completo de atualização de uma troca seja realizado em dois ciclos consecutivos. Com isso, estabelece-se uma sequência de execução do tipo $\text{Thread}_6 \rightarrow - \rightarrow \text{Thread}_5 \rightarrow - \rightarrow \ldots$, o que garante que as duas operações de escrita sejam realizadas sem violar as restrições de acesso impostas pela memória.

De forma específica, no primeiro ciclo, é executada a operação $N_2C(b) \leftarrow C_a$, enquanto no ciclo subsequente é realizada a operação $N_2C(a) \leftarrow C_b$. O mesmo procedimento é aplicado à estrutura C_2N, por meio das operações $C_2N(C_b) \leftarrow a$ e $C_2N(C_a) \leftarrow b$, respectivamente.

Além disso, os estágios iniciais do *pipeline* são reutilizados por cada *thread* em dois momentos distintos: inicialmente para a leitura dos dados e, posteriormente, para a escrita das atualizações. Por exemplo, enquanto a *thread*_9 realiza a leitura dos nós associados às células C_a e C_b no segundo estágio do *pipeline*, conforme ilustrado nas Figuras 3.1 e 3.3, a *thread*_2 pode, simultaneamente, executar a atualização dessas mesmas estruturas em sua região de memória privada. Como os dados associados a cada *thread* permanecem isolados, os acessos à memória ocorrem em endereços distintos, o que assegura a consistência das operações e elimina a possibilidade de condições de corrida durante a execução paralela.

3.2.7 Propagação dos dados

Durante a execução do algoritmo SA em *pipeline*, os dados associados a cada *thread* são propagados ao longo dos diferentes estágios do circuito. Informações como C_a e C_b, que representam as coordenadas das células envolvidas em uma possível troca, percorrem todos os estágios do *pipeline* até o estágio de cálculo de distâncias, no qual são utilizadas pela última vez. Em contraste, outros dados, como os identificadores dos nós a e b, são transmitidos apenas entre estágios consecutivos, conforme a necessidade de cada operação.

Entretanto, para a efetivação de uma troca aceita, torna-se necessário que as informações a, b, C_a e C_b estejam novamente disponíveis nos estágios iniciais do *pipeline*, a fim de permitir a atualização das estruturas de mapeamento N_2C e C_2N. Uma solução direta consistiria em propagar esses sinais por todo o *pipeline* até o estágio final e, em

seguida, retorná-los ao início. No entanto, essa abordagem implicaria a duplicação de trilhas de dados, no aumento do número de fios longos e, consequentemente, em maior complexidade e custo de roteamento do circuito.

Para evitar esse problema, foi adotado um mecanismo de *fila local* no segundo estágio do *pipeline*, responsável por armazenar as informações a, b, C_a e C_b durante a primeira passagem da *thread*. Essa estratégia foi denominada *pipeline vertical*, pois mantém os dados associados à atualização confinados a um único estágio do *pipeline*, de modo a evitar a necessidade de propagação ao longo de toda a estrutura. A fila local possui profundidade igual ao número de estágios do *pipeline*, o que garante que os dados permaneçam disponíveis até o instante correto de escrita nas estruturas de mapeamento.

Com essa organização, apenas um sinal binário, correspondente à decisão de aceitação ou rejeição da troca, necessita ser propagado do último estágio para o primeiro. Essa redução significativa no número de sinais em trânsito contribui para a diminuição da largura dos barramentos e para a simplificação do roteamento interno do circuito.

A Figura 3.3 evidencia dois aspectos centrais dessa abordagem. Inicialmente, as *threads* são inseridas no *pipeline* a cada dois ciclos de *clock*, de modo a permitir que a atualização das estruturas N_2C e C_2N seja realizada em dois ciclos distintos para que seja respeitada a restrição de uma única operação de escrita por ciclo imposta pelos blocos de memória. No primeiro ciclo, a posição associada ao nó b é atualizada com o valor C_a, enquanto os dados C_b e a são encaminhados ao estágio subsequente. No ciclo seguinte, quando a *thread* atinge o estágio posterior, realiza-se a atualização da posição associada ao nó a com o valor C_b.

Conforme ilustrado na Figura 3.3, enquanto a *thread*_9 percorre a fase de leitura das estruturas de mapeamento, a *thread*_2 executa a fase de escrita, em um esquema funcionalmente análogo ao estágio de *writeback* em arquiteturas RISC. Essa separação entre as fases de leitura e escrita garante a integridade dos dados, preserva a consistência das atualizações e permite a execução paralela de várias *threads*, sem a ocorrência de conflitos de acesso.

3.2.8 Adaptação para FPGA

As soluções apresentadas nesta e nas próximas seções fundamentam-se nos mesmos princípios estruturais discutidos na Seção 3.2.1, na qual foram desenvolvidas, implementadas e avaliadas arquiteturas paralelas organizadas em *pipeline* para a execução do algoritmo SA como posicionador de grafos de fluxo de dados (DFGs) em arquiteturas CGRAs. Nessa etapa anterior, foram exploradas estratégias de paralelismo temporal, particionamento de dados e replicação de estruturas de memória, com o objetivo de

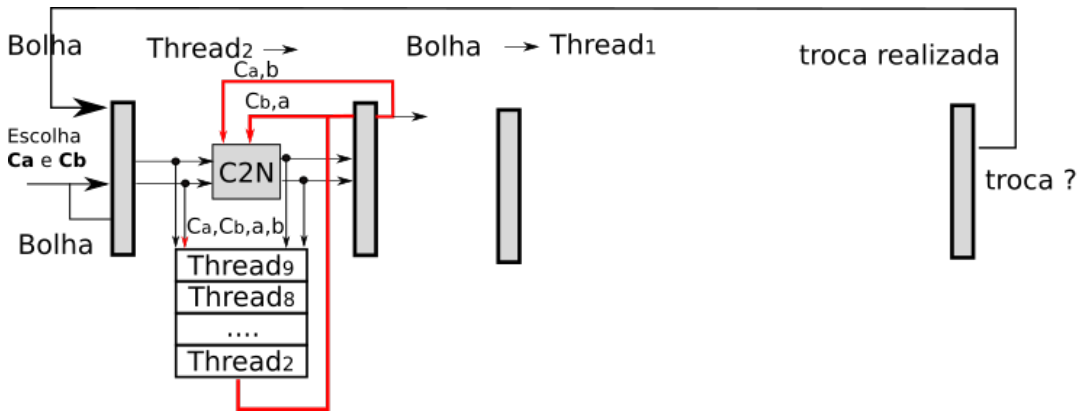


Figura 3.3: Atualização das memórias com múltiplas *threads* e redução de sinais.

maximizar o desempenho e a vazão do sistema.

Entretanto, a adaptação dessas arquiteturas para o domínio dos FPGAs impõe desafios adicionais que demandam modificações arquiteturais específicas. Tais desafios decorrem, principalmente, da elevada complexidade estrutural dos grafos extraídos de circuitos lógicos digitais sintetizados, bem como das restrições associadas ao consumo de recursos de memória embarcada e à necessidade de escalabilidade do sistema. Diferentemente dos CGRAs, nos quais os grafos tipicamente apresentam poucas dezenas de vértices, os grafos de circuitos em FPGAs podem conter milhares ou dezenas de milhares de nós, o que amplia significativamente a pressão sobre as estruturas de armazenamento e o acesso aos dados.

A principal modificação introduzida consiste na incorporação de mecanismos de *cache* voltados à otimização do acesso às estruturas de dados críticas, tais como as memórias de adjacência e os mapeamentos entre nós e células. Essa abordagem visa reduzir o consumo de blocos de memória embarcada (B-RAMs), além de possibilitar a replicação de múltiplas instâncias do *pipeline* no mesmo dispositivo FPGA para ampliar o grau de paralelismo.

3.2.9 *Pipeline* para o Algoritmo SA para FPGAs

O *pipeline* desenvolvido para o posicionamento em FPGAs preserva a estrutura funcional previamente apresentada, com a inserção de *threads* a cada dois ciclos de *clock*. Cabe destacar que, em função da adoção de mecanismos de *cache*, pode ocorrer a situação em que os dados requisitados por uma determinada *thread* não estão disponíveis imediatamente no momento do acesso. Nessas circunstâncias, torna-se necessária a consulta à memória principal, o que introduz latência adicional e resulta na formação de bolhas no *pipeline*, com *threads* temporariamente inativas. Para mitigar esse efeito e preservar uma elevada taxa de ocupação do *pipeline*, torna-se necessário inserir um número adicional de *threads*, de modo a mascarar a latência decorrente de acessos

fora da *cache*. Assim, as *threads* que retornam ao primeiro estágio do *pipeline* passam a ser organizadas em uma fila de espera, o que exige a implementação de mecanismos adicionais de controle para o correto escalonamento da execução.

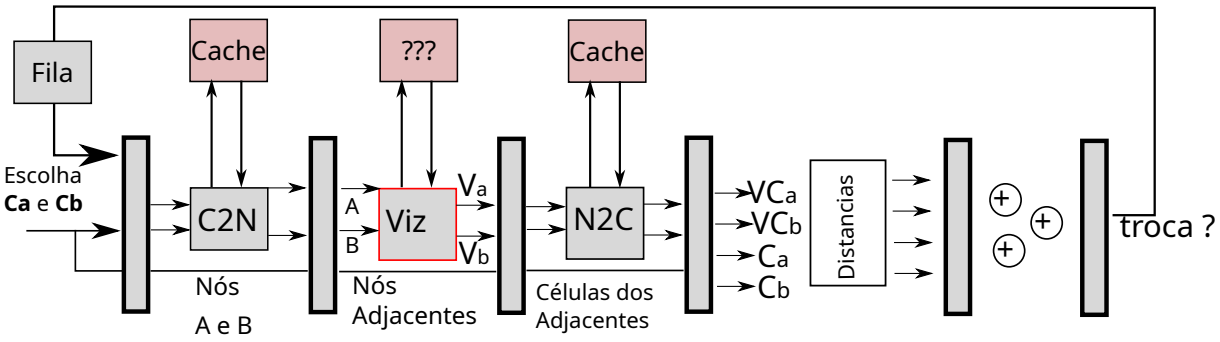


Figura 3.4: *Pipeline* simplificado Com modificação para acesso a *caches*.

Na Figura 3.4 são apresentadas as principais modificações estruturais introduzidas na arquitetura proposta. Os experimentos discutidos na Seção 4.2.4 foram conduzidos com a utilização de duas memórias *cache* independentes, destinadas ao armazenamento das estruturas N_2C (mapeamento de nó para célula) e C_2N (mapeamento de célula para nó).

Embora essa configuração inicial já proporcione uma redução significativa no consumo de blocos de memória embarcada (B-RAMs), outros cenários arquiteturais são considerados para trabalhos futuros. Entre as possibilidades, destacam-se a unificação das memórias *cache* em uma estrutura compartilhada e a ampliação do uso de *cache* para o armazenamento das listas de adjacência dos nós, conforme indicado na Figura 3.4 pelo estágio associado à memória denominada “Viz”.

3.2.10 Uso de Memória Cache

Os grafos extraídos de circuitos digitais sintetizados em FPGAs apresentam uma necessidade significativamente maior de módulos B-RAM. Isso ocorre porque a quantidade de células de posicionamento requeridas para grandes circuitos lógicos demanda vetores de tamanho proporcional tanto para representar os nós do grafo (N_2C) quanto para o mapeamento inverso (C_2N). Se essas estruturas forem replicadas integralmente para cada *thread* e o *pipeline* for replicado algumas vezes, o uso acumulado de B-RAMs pode ultrapassar a capacidade disponível do dispositivo. O cálculo de utilização está descrito na Seção 3.3.4.

Para contornar essa limitação, propôs-se a utilização de memórias cache locais, utilizadas para armazenar temporariamente os dados mais frequentemente acessados pelas *threads*. As estruturas N_2C e C_2N passam a ser acessadas inicialmente via cache, com acesso à memória principal apenas em caso de *cache miss*. Quando isso ocorre, a *thread* correspondente é suspensa e enfileirada até que os dados estejam disponíveis,

enquanto outras *threads* continuam a avançar no *pipeline*, de modo que seja possível mascarar a latência de acesso.

3.2.11 Desafios em QCA

Os algoritmos apresentados nesta seção fundamentam-se nos mesmos princípios estruturais explorados nas Seções 3.2.1 e 3.2.9, nas quais foram desenvolvidas, implementadas e avaliadas arquiteturas paralelas baseadas em *pipeline* para a execução do algoritmo SA aplicados às arquiteturas CGRAs e FPGAs. Entretanto, ao considerar a aplicação dessas abordagens ao contexto das arquiteturas baseadas em QCA, existem desafios específicos que demandam adaptações arquiteturais. Nesta tese desenvolveu-se apenas um esboço das adaptações para QCA, onde trabalhos futuros são necessários para avançar no tema.

Entre esses desafios, destacam-se o balanceamento de caminhos, as restrições de roteamento impostas pelos esquemas de *clocking* e a necessidade de evitar cruzamentos de sinais. Tais características diferenciam substancialmente o problema de posicionamento em QCA daqueles tratados nos contextos de CGRAs e FPGAs.

As arquiteturas apresentadas nas próximas Seções são adaptações das versões previamente desenvolvidas para CGRAs e FPGAs, com ênfase nas restrições de roteamento decorrentes dos esquemas de *clocking* USE e *2DDWave*, de modo a assegurar a validade funcional dos circuitos posicionados em QCA.

3.2.12 Proposta de *Pipeline* para o SA em QCAs

Primeiramente, assume-se que o grafo apresenta um mapeamento um-para-um com a arquitetura-alvo, o que permite tratar simultaneamente os problemas de posicionamento e de roteamento. Além das portas lógicas, o grafo inclui vértices que representam segmentos de conexão (fios). Portanto, o estágio de avaliação de custo é substituído por uma lógica baseada em penalidades e bonificações associadas à correção topológica das conexões, conforme definido na função de custo proposta nesta tese. Essa modificação reduz a complexidade do circuito, elimina a necessidade de unidades aritméticas dedicadas ao cálculo explícito de distâncias e preserva a capacidade do algoritmo de explorar o espaço de soluções de forma estocástica.

Para o algoritmo SA, o circuito previamente apresentado na Seção 3.2.1 pode ser reutilizado, desde que sejam realizadas adaptações específicas no mecanismo de avaliação da posição relativa entre os nós selecionados para a troca. Diferentemente das arquiteturas CGRAs e FPGAs, nas quais a métrica de distância de *Manhattan* foi aplicada, o mapeamento simultâneo com o posicionamento e roteamento no QCA simplifica essa abordagem, uma vez que as células possíveis para cada nó adjacente são pré-definidas pelo esquema de *clocking* utilizado.

No *USE* e no *2DDWave*, cada célula QCA possui exatamente duas portas de entrada e duas de saída (exceto as que se encontram nas bordas), cuja orientação depende da linha e da coluna em que a célula está localizada na matriz de posicionamento. Essa característica permite determinar quais células são candidatas válidas para a alocação dos nós adjacentes, com base na posição da célula de origem e na direção do fluxo de sinal.

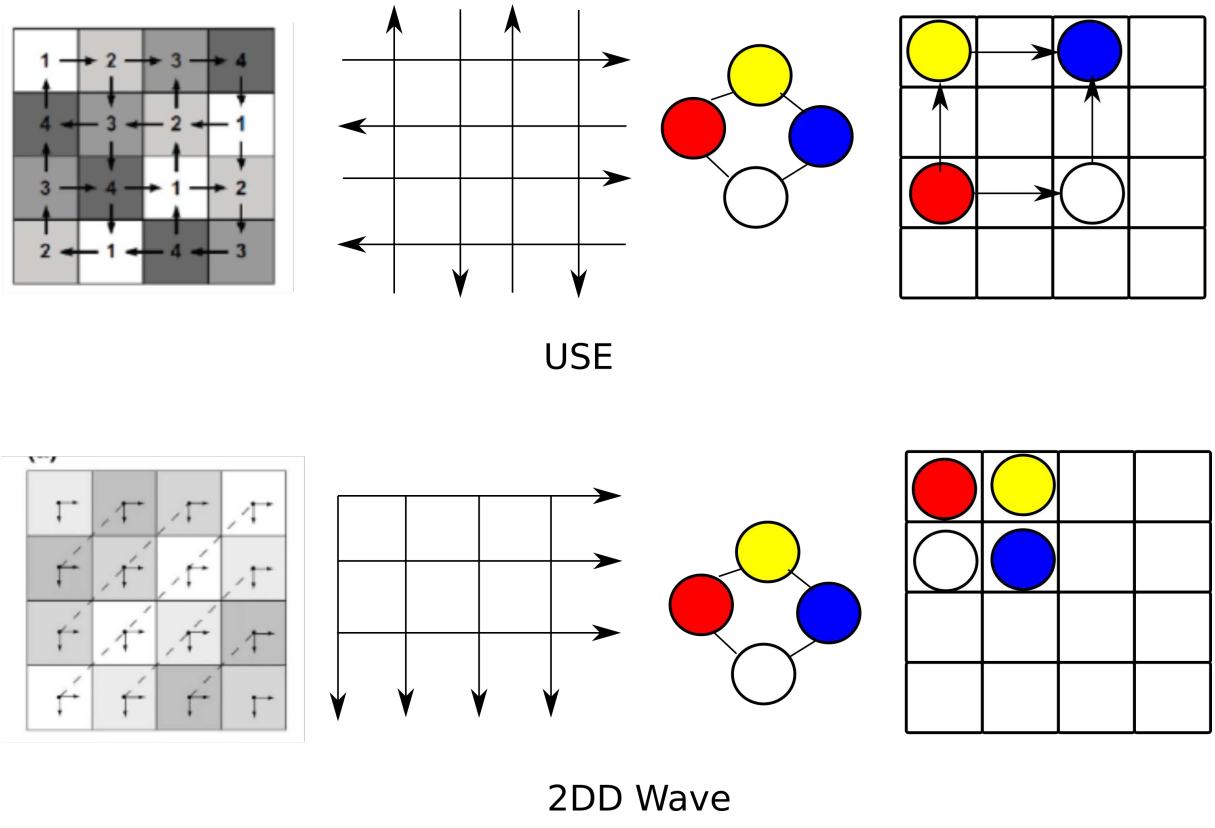


Figura 3.5: Esquema de *clocking* USE e *2DDWave*

A Figura 3.5 ilustra o padrão recorrente de orientação das células ao longo da matriz de posicionamento nos esquemas de *clocking* USE e *2DDWave*. Observa-se, por exemplo, que a célula localizada na posição $(0,0)$ apresenta a mesma configuração estrutural daquela em $(2,2)$; isso mostra a regularidade espacial determinada pelas coordenadas da matriz. Essa propriedade permite inferir a orientação de qualquer célula a partir da configuração de referência da célula inicial $(0,0)$, devido às simetrias presentes no arranjo.

No exemplo ilustrado na figura, o mapeamento do grafo exige a mudança de colunas para viabilizar o roteamento das conexões, em função das restrições impostas pela orientação das células no esquema USE. Tal característica deixa claro a influência direta do esquema de *clocking* na viabilidade das conexões físicas entre os nós do circuito.

No caso da arquitetura *2DDWave*, também apresentada na Figura 3.5, o mesmo exemplo admite uma implementação mais direta e se mostra mais adequada à propa-

gação de sinais ao longo de trajetórias diagonais em circuitos combinacionais. Nessa arquitetura, cada célula dispõe de duas portas de entrada posicionadas nas direções superior e esquerda, e duas portas de saída orientadas para a direita e para baixo. Essa padronização simplifica significativamente a verificação de conexões válidas entre células adjacentes, uma vez que a definição das direções de entrada e saída é determinada de forma explícita pela posição da célula na matriz.

A regularidade estrutural das arquiteturas USE e *2DDWave*, aliada à função de custo proposta na Seção 3.2.13, permite uma simplificação do *pipeline* implementado para o algoritmo SA. Em particular, elimina-se a necessidade de cálculos explícitos de distância entre nós conectados. Para cada tentativa de troca, torna-se suficiente verificar se os nós adjacentes ao nó selecionado estão posicionados nas células previamente estabelecidas como válidas pelo esquema de *clocking*. Essa verificação é realizada por meio de acessos à memória N_2C, responsável pelo mapeamento entre nós e posições na matriz, e pela aplicação direta da função de custo definida para o contexto de QCA.

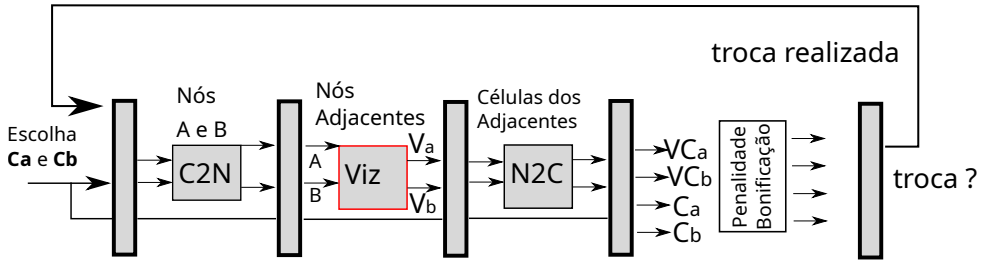


Figura 3.6: Proposta de *pipeline* para o SA para QCAs.

A Figura 3.6 apresenta a principal modificação feita na arquitetura do circuito para a execução do algoritmo SA para o QCA, a qual se concentra no estágio de verificação da função de custo. Diferentemente das abordagens empregadas nas arquiteturas FPGA e CGRA, nas quais o custo de uma troca é obtido a partir do cálculo explícito da distância total entre os nós conectados, o modelo proposto para QCA adota uma estratégia simplificada. Nessa abordagem, a decisão de aceitar ou rejeitar uma troca baseia-se exclusivamente em um sistema de bonificações e penalidades associadas à correção topológica do posicionamento dos nós.

Essa modificação reduz a complexidade do circuito, uma vez que elimina a necessidade de unidades aritméticas dedicadas ao cálculo de distâncias e à acumulação de custos parciais. Um segundo fator relevante de simplificação está relacionado à conectividade limitada dos nós no modelo QCA. Cada elemento pode apresentar, no máximo, quatro conexões, correspondentes a até duas entradas e duas saídas. Essa restrição permite que a largura da memória de adjacência (*Viz*) seja fixa e previamente definida, o que simplifica tanto o projeto da memória quanto a lógica de controle associada.

Para circuitos de maior porte, pode-se ainda estender essa arquitetura por meio da

utilização de memórias *cache* para as estruturas C_2N, N_2C e *Viz*, de forma análoga ao que foi apresentado na Seção 3.2.9. Essa adaptação tem como objetivo a redução do consumo de memória embarcada e a viabilização da escalabilidade do posicionamento em matrizes de maior dimensão, sem o comprometimento da organização do *pipeline* ou a consistência das atualizações.

3.2.13 Função de Custo

O algoritmo *Simulated Annealing* (SA) utiliza uma função de custo como critério para avaliar a qualidade relativa de diferentes soluções de posicionamento. Essa função quantifica o impacto de uma possível troca entre dois nós do grafo e, a partir dessa avaliação, define a probabilidade de aceitação da troca, inclusive nos casos em que a modificação resulta em uma degradação local da solução.

Esse mecanismo probabilístico é um elemento central do algoritmo, pois permite escapar de mínimos locais e favorece uma exploração mais ampla do espaço de busca. Ao admitir, de forma controlada, soluções temporariamente piores, o SA amplia a diversidade das configurações avaliadas e aumenta a probabilidade de convergência para soluções de melhor qualidade global.

A probabilidade de aceitação de uma troca é definida pela seguinte expressão:

$$P = \exp\left(\frac{-\Delta}{T}\right),$$

onde Δ representa a diferença entre o custo da solução após a aplicação da troca e o custo da solução corrente, enquanto T denota a temperatura atual do sistema. O valor de Δ é definido pela expressão:

$$\Delta = Custo_{futuro} - Custo_{presente}$$

Quando $\Delta \leq 0$, a troca é aceita de forma determinística, uma vez que essa condição implica que o custo da solução resultante é inferior ou igual ao custo da solução corrente. Trata-se, portanto, de uma condição favorável à troca. Quando $\Delta > 0$, a probabilidade de aceitação decresce exponencialmente à medida que o valor de Δ aumenta. O parâmetro T exerce influência direta sobre essa probabilidade: valores elevados de temperatura resultam em maiores probabilidades de aceitação, enquanto valores reduzidos tornam o critério progressivamente mais restritivo. Nas etapas iniciais do algoritmo, caracterizadas por temperaturas elevadas ($T \gg 1$), as trocas que acarretam degradações significativas ainda apresentam uma probabilidade não desprezível de aceitação. À medida que o sistema é resfriado ($T \ll 1$), essa permissividade diminui, e o algoritmo passa a privilegiar soluções de melhor qualidade.

Para o posicionamento em arquiteturas CGRAs e FPGAs, adota-se uma função de

custo baseada na distância *Manhattan* entre nós conectados. Essa métrica mostra-se adequada em ambos os domínios, ainda que por motivações distintas. No contexto de CGRAs, a minimização das distâncias tende a reduzir o desbalanceamento temporal entre caminhos de dados, contribuindo para a diminuição da profundidade das filas necessárias ao reequilíbrio do grafo posicionado. Contudo, tal redução não é garantida, uma vez que o balanceamento temporal não é avaliado explicitamente durante a etapa de posicionamento. Nesta tese, a métrica de custo adotada no posicionamento visa exclusivamente à redução do custo das arestas, enquanto a avaliação final das soluções considera, de forma separada, tanto o número de interconexões quanto a qualidade do balanceamento temporal, o qual é verificado apenas após a etapa completa de roteamento.

Em arquiteturas FPGAs, a redução da distância de *Manhattan* pode impactar diretamente o comprimento do caminho crítico, o congestionamento das trilhas de roteamento e favorecer a temporização e a viabilidade do roteamento físico do circuito. Contudo, a minimização dessa distância não assegura, por si só, a obtenção do menor caminho crítico, uma vez que fatores adicionais, como a disponibilidade de recursos de interconexão e a distribuição das conexões, influenciam no atraso final do circuito.

Entretanto, essa abordagem não se mostra adequada para o posicionamento em arquiteturas baseadas em QCA, nas quais as células apresentam conectividade física e direcionalidade previamente definidas.

Diante dessa limitação, diferentes versões da função de custo foram propostas e avaliadas empiricamente com o objetivo de identificar critérios capazes de refletir de forma mais fiel a qualidade do posicionamento em QCA. A primeira versão baseou-se exclusivamente na contagem do número de nós corretamente posicionados. Contudo, essa formulação apresentou baixa sensibilidade a modificações locais, uma vez que o valor de Δ dependia apenas da quantidade total de nós corretamente posicionados, independentemente de melhorias ou degradações estruturais decorrentes de uma troca específica.

Na segunda versão foi incorporada a soma das distâncias entre os nós e seus respectivos vizinhos. Esse acréscimo aumentou a resolução da função de custo e produziu maior variabilidade entre estados sucessivos. Ainda assim, essa formulação passou a favorecer configurações em que os nós adjacentes se encontravam espacialmente próximos, mesmo quando dispostos de forma incompatível com a orientação direcional imposta pela topologia da arquitetura.

A terceira versão da função de custo adotou um modelo baseado em penalizações e bonificações. Nessa formulação, cada nó corretamente posicionado contribui com um bônus unitário positivo ao custo final, enquanto nós posicionados de forma inadequada ou que apresentem conexões com orientação invertida recebem uma penalidade de três unidades. Adicionalmente, quando os dois nós selecionados para uma troca são

adjacentes, introduz-se uma penalidade adicional de duas unidades, com o objetivo de reduzir a probabilidade de inversão direta de conexões entre eles. A função de custo passa, assim, a ser definida como o somatório das penalidades associadas à posição e à inversão, subtraído do total de bonificações obtidas pelos acertos.

Após esse somatório, o valor resultante é normalizado em relação à penalidade máxima que os nós adjacentes aos nós selecionados para a troca poderiam gerar. Por exemplo, considerando um nó A com dois nós adjacentes, a penalidade máxima associada corresponde a $penalidade_{\max} = 2 \times 3$, isto é, o número de nós adjacentes multiplicado pela penalidade máxima por nó. Com essa normalização, os custos associados às soluções corrente e futura assumem valores no intervalo $[0, 1]$. Essa terceira formulação foi adotada para a obtenção dos resultados experimentais apresentados na Seção 4.3 do Capítulo 4.

3.3 Implementações dos algoritmos YOTO e YOTT em Arquiteturas Reconfiguráveis

A partir desta seção, o *pipeline* desenvolvido para o algoritmo SA é estendido para a implementação dos algoritmos YOTO e YOTT. Essa extensão justifica-se pelo fato de que ambos apresentam uma estrutura iterativa com dependências de dados entre iterações consecutivas, o que restringe a paralelização direta, conforme discutido na Seção 2.3. Assim como no SA, o estado produzido em uma iteração influencia diretamente as decisões da iteração seguinte. Nesse contexto, a organização em *pipeline* constitui um mecanismo adequado para explorar o paralelismo temporal por meio da execução sobreposta de múltiplas instâncias independentes.

A execução concorrente de múltiplas *threads* impõe desafios relacionados ao compartilhamento e à movimentação eficiente de dados ao longo do *pipeline*. A solução para o desenvolvimento da arquitetura deve evitar gargalos de leitura, minimizar a contenção nas estruturas de memória e restringir a introdução de recursos adicionais que elevem a complexidade do circuito e dificultem os processos de síntese e roteamento em FPGA. Do ponto de vista metodológico, as próximas Seções adotam um modelo de arquitetura em comum, reutilizado entre diferentes domínios e algoritmos, e concentram-se nas adaptações dos estágios diretamente afetados pelas restrições da arquitetura-alvo (CGRA, FPGA ou QCA) e pela política de alocação empregada (YOTO ou YOTT).

3.3.1 *Pipeline* para o Algoritmo YOTO para CGRAs

A Figura 3.7 apresenta a estrutura em *pipeline* com suporte a SMT implementada para o algoritmo YOTO, descrito na Seção 2.3.1. Nessa arquitetura, cada *thread* ativa mantém um ponteiro de aresta, responsável por indicar qual vértice do grafo deve ser alocado na próxima etapa do posicionamento na matriz do CGRA.

Trata-se de uma abordagem incremental, na qual cada iteração processa uma única aresta para realizar o posicionamento de um vértice. Em situações nas quais a célula candidata encontra-se ocupada, a mesma aresta pode percorrer o *pipeline* novamente até que uma posição válida seja identificada. Ainda assim, essa estratégia apresenta uma vantagem em relação ao algoritmo SA, uma vez que não exige a inicialização prévia da memória com um posicionamento aleatório.

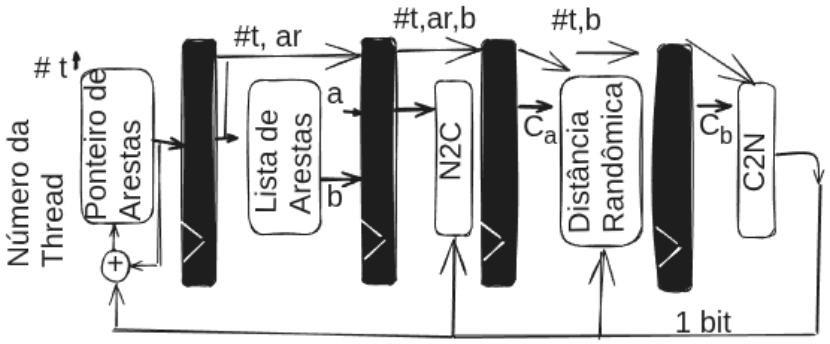


Figura 3.7: Projeto para o *pipeline* para o algoritmo YOTO.

No segundo estágio do *pipeline*, os nós a e b são identificados a partir do identificador da *thread* e do ponteiro de aresta associado. No terceiro estágio, determina-se a posição atual C_a do nó a na matriz de posicionamento. O quarto estágio executa uma iteração do laço interno responsável por identificar uma célula adjacente C_b a partir de C_a, de acordo com os critérios de vizinhança definidos pelo algoritmo. No estágio final, verifica-se a disponibilidade da célula C_b para alocação. Caso a célula esteja livre, o nó b é alocado, e três atualizações são realizadas: a associação da célula C_b ao nó b na memória C_2N, o mapeamento do nó b para a célula C_b na memória N_2C, e o incremento do ponteiro de arestas da *thread*, que passa a indicar a próxima aresta da lista de travessia.

Dessa forma, à medida que cada *thread* percorre os estágios do *pipeline*, o algoritmo tenta realizar o posicionamento do nó b em uma célula adjacente válida, conforme a ordem de travessia do grafo. O ponteiro de arestas exerce função análoga ao contador de programa (PC) em arquiteturas *multithread*, enquanto a lista de arestas assume papel equivalente a um fluxo de instruções. Assim, cada aresta pode ser interpretada como uma instrução processada sequencialmente pelo algoritmo, com atualização explícita do ponteiro a cada alocação bem-sucedida.

Em contraste com a implementação do algoritmo SA, que exige múltiplas operações simultâneas de leitura e escrita em memórias compartilhadas e apresenta dependências de dados mais intensas entre os estágios, a arquitetura em *pipeline* proposta para o algoritmo YOTO adota uma lógica de controle mais simples e demanda um número reduzido de acessos concorrentes às estruturas de memória. Ainda assim, o emprego de mecanismos de escolha pseudoaleatória de células adjacentes preserva o comportamento estocástico do algoritmo, de modo a promover a diversidade no espaço de soluções e evitar padrões determinísticos de alocação.

Cabe ressaltar que a estrutura fundamental dessa arquitetura de travessia mantém os mesmos princípios adotados no acelerador do algoritmo SA. Embora o algoritmo de posicionamento seja distinto, permanecem válidas as estratégias de utilização de memórias locais e independentes por estágio, a execução concorrente de múltiplas *threads* para mascarar latências e a replicação controlada das unidades de processamento. Dessa forma, a arquitetura proposta configura-se como um modelo genérico, passível de adaptação a diferentes algoritmos de posicionamento.

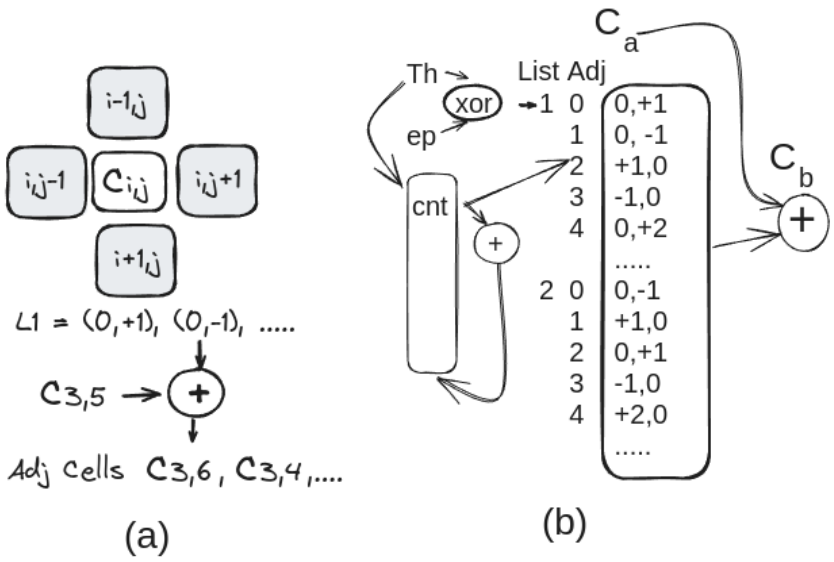


Figura 3.8: (a) Lista de índices das células adjacentes; (b) Gerador pseudo-randômico para a busca por células adjacentes.

A Figura 3.8 (a) ilustra o modelo do grafo da matriz empregado na busca por células adjacentes durante o processo de posicionamento. A partir de uma célula localizada nas coordenadas (i, j), as células vizinhas são obtidas por meio de deslocamentos unitários nas direções horizontal e vertical. Em particular, os deslocamentos $(0, +1)$, $(0, -1)$, $(-1, 0)$ e $(+1, 0)$ correspondem, respectivamente, às direções leste, oeste, norte e sul em relação à posição (i, j).

Com o objetivo de preservar o caráter estocástico da busca por células livres, foram previamente construídas múltiplas listas que contêm as direções possíveis em ordens

distintas. Por exemplo, uma lista pode priorizar a verificação nas direções oeste, norte, leste e sul, enquanto outra pode iniciar pelas direções leste, oeste, norte e sul. A seleção da lista utilizada por cada *thread* baseia-se em um mecanismo de pseudoaleatoriedade definido por uma operação lógica de OU-exclusivo (`XOR`) entre o identificador da *thread* e o ponteiro de arestas associado. Essa estratégia induz variação entre as instâncias concorrentes e contribui para a diversidade das rotas de alocação exploradas.

Cada *thread* mantém um contador local que registra o número de tentativas de alocação já realizadas para uma dada célula, o que permite a varredura progressiva das opções disponíveis na lista selecionada. As listas são organizadas de forma hierárquica e priorizam inicialmente as células a uma distância de *Manhattan* igual a 1, seguidas por aquelas a uma distância de 2 e, assim, sucessivamente. Essa organização preserva a localidade espacial da busca e favorece a obtenção de soluções de melhor qualidade.

A Figura 3.8 (b) apresenta o padrão de travessia resultante dessa estratégia, com destaque para a combinação entre proximidade espacial e pseudoaleatoriedade como princípio orientador da busca por células livres no grafo da matriz. A ordem de travessia das arestas do grafo é definida por um algoritmo de busca em profundidade (DFS), conforme descrito em [16].

3.3.2 *Pipeline* para o Algoritmo YOTT para CGRAs

A implementação do algoritmo YOTT em *pipeline* é basicamente uma extensão da arquitetura previamente desenvolvida para o YOTO. Para esse fim, foram incorporados novos elementos funcionais ao circuito, com destaque para a lógica de verificação das anotações associadas às arestas e para o mecanismo de seleção da célula de destino C_b.

Diferentemente do YOTO, no qual a única condição para a alocação consiste na disponibilidade da célula candidata, o YOTT impõe, adicionalmente, o atendimento às restrições de distância definidas pelas anotações da aresta em processamento, conforme descrito na Seção 2.3.2. A Figura 3.9 apresenta as principais modificações introduzidas na arquitetura para suportar essa ampliação funcional.

Cada aresta pode conter até três anotações, compostas por nós de referência e suas respectivas distâncias. Durante a execução, essas informações são propagadas ao longo do *pipeline* e permanecem associadas à *thread* correspondente. No terceiro estágio, os nós anotados são convertidos em coordenadas de posição na matriz de células, identificadas como C_1, C_2 e C_3.

Para os casos em que não existem anotações associadas à aresta, um código especial é propagado juntamente com os demais dados, de modo a assegurar que a ausência de informações não interfira nas etapas subsequentes de verificação. Um novo estágio, denominado *verificação de restrições*, é introduzido no *pipeline* com o objetivo de validar se a célula candidata C_b satisfaz integralmente as exigências impostas pelas anotações

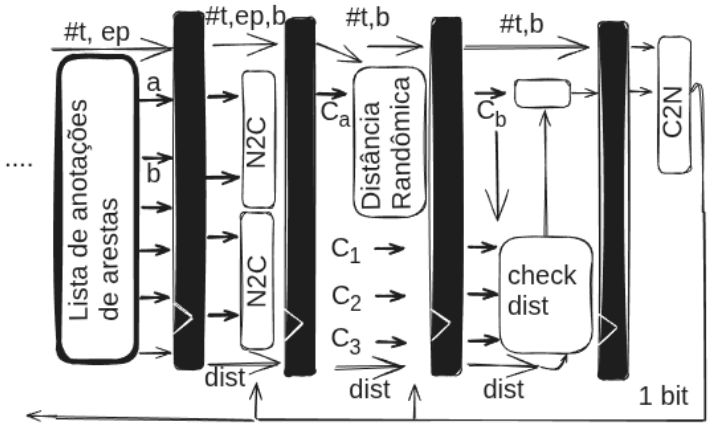


Figura 3.9: Estágio de anotações das arestas para o algoritmo YOTT *pipeline* com o estágio de verificação da anotação.

de distância. Quando as restrições não são plenamente atendidas, o circuito seleciona, dentre as alternativas disponíveis, a célula que melhor se aproxima das condições estabelecidas.

Enquanto não ocorre a identificação de uma célula válida, a *thread* prossegue na exploração do conjunto de células adjacentes por meio de listas de deslocamentos geradas de forma pseudoaleatória. Esse processo estende-se até o atendimento das restrições ou até o alcance de um número máximo de tentativas. Com o intuito de limitar o consumo de ciclos computacionais, o módulo de verificação adota um limite de tentativas configurável. Ultrapassado esse limite, o circuito força a aceitação da célula que apresenta a menor violação das restrições e retorna um sinal de validação juntamente com as coordenadas da célula selecionada. O algoritmo empregado para a geração da lista de arestas segue a estratégia *zigzag*, conforme descrito em [15].

3.3.3 *Pipeline* para YOTO e YOTT em FPGAs com Cache

Nos FPGAs, as arquiteturas para YOTO e YOTT reutilizam o *pipeline* proposto para CGRAs, com adaptações concentradas no armazenamento e no acesso aos dados de posicionamento, de modo análogo às modificações introduzidas para o SA. O foco recai sobre o acesso às estruturas N_2C e C_2N, que passam a ser mediados por memórias *cache* para reduzir o consumo de B-RAM e viabilizar a escalabilidade.

Assim como no domínio CGRA, no YOTO cada *thread* mantém um ponteiro local que indica a aresta a ser processada, e o grafo é percorrido segundo uma ordem definida por uma estratégia de travessia (por exemplo, DFS). Em cada ciclo, o *pipeline* identifica os nós a e b, localiza a célula C_a do nó já alocado e inicia a busca por uma célula C_b adjacente e livre. A busca utiliza listas embaralhadas de direções, indexadas por `XOR` entre o ponteiro de aresta e o identificador da *thread*, o que preserva a diversidade espacial entre instâncias concorrentes.

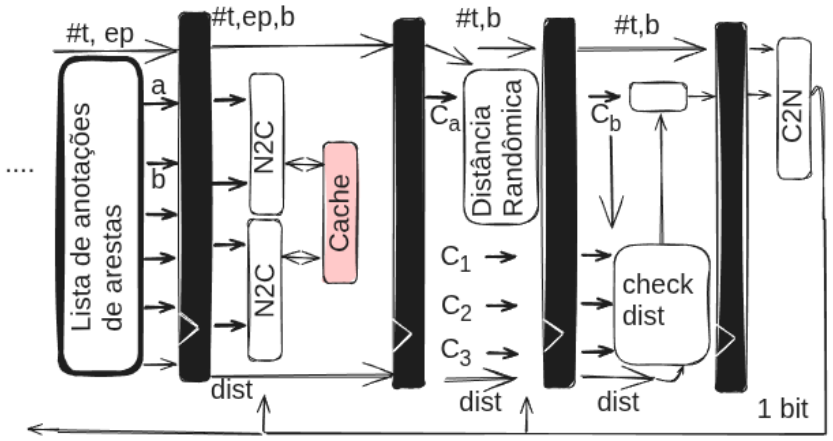


Figura 3.10: YOTO em *pipeline* com acesso a memórias *cache*.

As estruturas N_2C e C_2N são acessadas pelo *cache*. Em cada acesso, verifica-se a presença do dado no *cache*. Em caso de ausência (*cache miss*), a *thread* é suspensa e enfileirada até que o dado seja recuperado da memória principal. Essa política permite mascarar a latência da memória externa por meio de múltiplas *threads*, o que preserva a ocupação do *pipeline* em níveis elevados, de modo coerente com a estratégia adotada no SA.

O YOTT estende essa organização ao incorporar a verificação de restrições topológicas derivadas das anotações associadas às arestas. Cada aresta pode conter até três anotações que representam as distâncias mínimas exigidas entre o nó em alocação e os nós previamente posicionados. No *pipeline*, essas anotações são convertidas em coordenadas de referência e avaliadas no estágio de verificação de restrições, que valida se C_b atende aos limites estabelecidos.

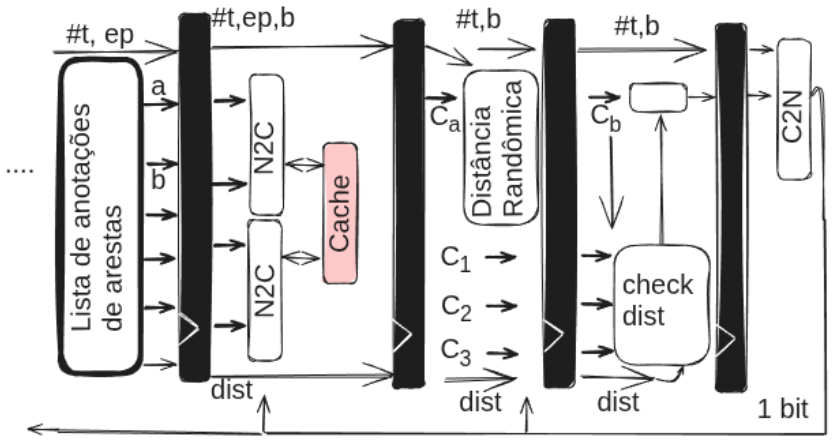


Figura 3.11: Estágio de anotações e verificação de restrições no YOTT com acesso a *cache*.

Na ausência de células válidas, a *thread* mantém a busca pseudoaleatória até atingir um limite de tentativas. Caso nenhuma célula satisfaça integralmente as restrições,

a alocação recai sobre a melhor célula dentro dos limites definidos. A Figura 3.11 evidencia a principal alteração no estágio de anotações: a introdução do acesso via *cache* não apenas para C_2N, mas também para N_2C.

Para garantir escalabilidade, a arquitetura utiliza múltiplos *caches* locais, distribuídos por estágios, para reduzir a pressão sobre B-RAM e suportar grafos com dezenas de milhares de nós e arestas. Adicionalmente, o *pipeline* admite a parametrização do número de *threads* e da profundidade, de modo a ajustar o mascaramento de latência conforme a taxa de falhas em *cache*.

3.3.4 Utilização de Recursos B-RAM para os Aceleradores

Para ilustrar o problema de utilização de recursos, toma-se como referência o *benchmark div* (Tabela 4.10), do conjunto EPFL [1], com mais de 10.000 nós e aproximadamente 30.000 arestas. Mesmo com uma única *thread*, o algoritmo requer vetores com milhares de posições para representar C_2N e N_2C. A replicação integral dessas estruturas por *thread*, conforme a estratégia típica no domínio CGRA, eleva o consumo de B-RAM e pode inviabilizar múltiplas instâncias.

Como base, considera-se o FPGA Xilinx VU9P, que possui aproximadamente 2,5 milhões de LUTs, 6800 DSPs e 2160 blocos B-RAM com um total de cerca de 7,9 MB de armazenamento interno [99]. Para o circuito *div*, a matriz possui aproximadamente 100×100 posições (considerando apenas a quantidade de nós). Assim, 14 b endereçam até $2^{14} = 16.384$ posições, e outros 14 b codificam o identificador do nó associado. Obtém-se, portanto, 28 b por vértice/posição.

Dessa forma, as estruturas N_2C e C_2N demandam, no mínimo, 28 blocos B-RAM (com capacidade aproximada de 30 Kb) para armazenar as 10.000 posições do circuito. Para execução com pelo menos dez *threads* independentes, o armazenamento escala para $10 \times 10.000 = 100.000$ posições em cada memória. Esse volume corresponde a $\frac{100.000}{30.000} \approx 4$ blocos por conjunto de 28 b e, portanto, $4 \times 28 \approx 112$ blocos para as duas memórias, ou 56 blocos para cada uma.

Adicionalmente, a lista de adjacências (*Viz*) deve ser considerada. Assumindo até quatro conexões por nó, a memória requer aproximadamente $4 \times 14 = 56$ blocos B-RAM para armazenar identificadores de adjacência em uma matriz com 10.000 elementos.

No SA em *pipeline*, a memória N_2C requer replicação v vezes, onde v corresponde ao número máximo de vizinhos, pois múltiplas leituras simultâneas ocorrem em um mesmo estágio. Para $v = 4$, obtêm-se quatro instâncias independentes de N_2C, o que adiciona $4 \times 56 = 224$ blocos B-RAM. Somando-se 56 blocos para C_2N e 56 blocos para *Viz*, uma única instância do circuito com dez *threads* consome, no mínimo, 336 blocos B-RAM, aproximadamente 15% do total disponível. Para múltiplas cópias do *pipeline*,

o consumo cresce proporcionalmente e pode exceder 90% com cerca de seis cópias.

Para YOTO e YOTT, o consumo segue metodologia semelhante, sendo a arquitetura composta por 10 *threads* por instância.. As estruturas N_2C e C_2N permanecem e requerem aproximadamente 56 blocos cada, totalizando 112 blocos.

Ao contrário do SA, YOTO e YOTT utilizam estrutura explícita de arestas. Cada aresta armazena um par (u, v), com 14 b por identificador, totalizando 28 b por aresta. Para 30.000 arestas, a estrutura demanda aproximadamente 28 blocos B-RAM, o que resulta em $112 + 28 = 140$ blocos para o YOTO, cerca de 7% da capacidade total.

O YOTT acrescenta anotações por aresta. Considerando até três anotações, com 14 b para o índice do nó e 8 b para a distância, cada anotação utiliza 22 b; logo, $3 \times 22 = 66$ b adicionais por aresta. Para essa estrutura, obtêm-se 66 blocos adicionais. Assim, o consumo estimado para YOTT totaliza 112 blocos (N_2C e C_2N) + 28 blocos (arestas) + 66 blocos (anotações) + 56 blocos (replicação de N_2C), o que resulta em 262 blocos B-RAM, aproximadamente 12% do total. Nessas condições, dez cópias de YOTO e oito de YOTT totalizam, respectivamente, 70% e 96% dos blocos disponíveis. A Tabela 3.1 resume os valores.

Tabela 3.1: Número mínimo de blocos B-RAMs para uma unidade em *hardware* dos algoritmos SA, YOTO e YOTT.

Algoritmo	**C2N**	**N2C**	**Viz**	**Arestas**	**Anot.**	**Total (%)**
SA	56	224	56	-	-	336 (15%)
YOTO	56	56	-	28	-	140 (7%)
YOTT	56	112	-	28	66	262 (12%)

Como alternativa para reduzir o consumo de B-RAMs, a arquitetura adota um *cache* de tamanho fixo para armazenar temporariamente os dados mais acessados. Essa estratégia reduz a quantidade de blocos necessária por instância e viabiliza uma maior replicação do acelerador no dispositivo.

3.3.5 Busca em Largura Local para YOTO e YOTT em FPGAs

O procedimento de busca por células livres empregado pelos algoritmos YOTO e YOTT baseia-se em uma varredura em largura a partir da célula corrente, na qual se encontra o nó de origem da aresta considerada. Dessa forma, a seleção de posições candidatas prioriza, em ordem crescente, as células com menor distância *Manhattan*, conforme discutido na Seção 2.3.

Por se tratar de uma heurística gulosa, esse mecanismo de busca [32] tende a exigir múltiplas execuções em circuitos com mais de 1.000 LUTs, especialmente quando a área total é restringida ao quadrado mínimo capaz de acomodar o circuito. Como exemplo, considere um grafo com 4096 LUTs mapeado em uma matriz de 64×64

LUTs. Em razão do caráter incremental do posicionamento e do elevado grau de conectividade dos vértices, regiões da matriz tornam-se rapidamente densamente ocupadas, o que dificulta a identificação de células livres na vizinhança próximas aos vértices já posicionados.

Embora uma execução do algoritmo DFS apresente um tempo de execução aproximadamente 1.000 vezes inferior ao do VPR 5.0 [57], conforme descrito na Seção 4.2, a necessidade de repetições reduz significativamente essa vantagem. Além disso, o número médio de tentativas cresce de forma acentuada em circuitos com milhares de LUTs, o que limita a escalabilidade prática do método.

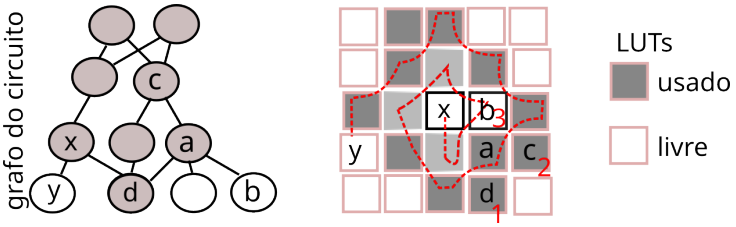


Figura 3.12: Dois exemplos de posicionamento para as arestas $a - b$ e $x - y$.

O princípio da estratégia é a transferência de localidade, que orienta o posicionamento incremental por meio da proximidade espacial entre vértices adjacentes. A Figura 3.12 exemplifica esse comportamento: ao processar a aresta $a - -b$, o nó a já se encontra alocado e a busca em largura identifica uma célula livre nas proximidades para posicionar b em poucos passos. Entretanto, em estágios avançados do posicionamento, quando a matriz apresenta elevada taxa de ocupação, a mesma estratégia pode demandar um número substancialmente maior de verificações até a identificação de uma célula disponível. No exemplo da aresta $x - -y$ da Figura 3.12, torna-se necessário inspecionar ao menos 13 posições candidatas para localizar uma célula livre. Esse aumento no esforço de busca afeta diretamente circuitos grandes e densos, nos quais a disponibilidade de células livres nas proximidades é reduzida e a minimização de área intensifica a pressão sobre o processo de posicionamento.

Em circuitos digitais mapeados em FPGAs, a quantidade de nós adjacentes a um determinado vértice pode ser significativamente maior do que nas abordagens voltadas para arquiteturas CGRA e QCA. Embora uma LUT de seis entradas imponha um limite fixo ao número de sinais de entrada, o número de saídas associadas a um vértice é, na prática, virtualmente ilimitado. Essa característica resulta em grafos com elevado grau de *fan-out*, o que impacta diretamente o comportamento dos algoritmos YOTO e YOTT.

Como os algoritmos adotam uma estratégia incremental de posicionamento que prioriza a alocação de células próximas ao nó de origem da aresta considerada, à medida que o *fan-out* aumenta, as células mais próximas tendem a ser rapidamente

ocupadas, não apenas pelo próprio nó de origem, mas também por outros vértices pertencentes a regiões densamente conectadas do grafo. Como consequência, a busca por células livres nas proximidades torna-se progressivamente mais complexa.

Ao buscar alta densidade de ocupação de LUTs com minimização de área, os algoritmos apresentam bom desempenho nas fases iniciais do posicionamento, quando a maioria das células ainda se encontra disponível e a alocação ocorre de forma direta. Entretanto, com o aumento da taxa de ocupação, a identificação de posições livres passa a exigir um esforço crescente de busca, o que pode impactar significativamente o tempo total de execução, sobretudo nas etapas finais do posicionamento.

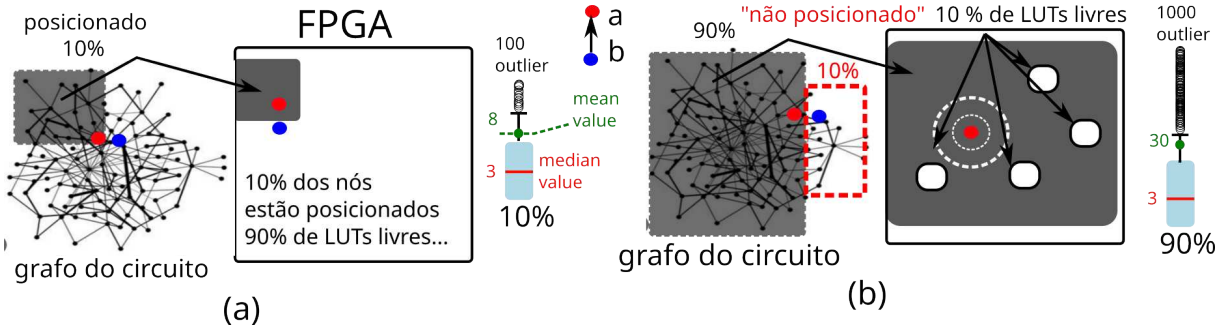


Figura 3.13: Posicionamento incremental: (a) Cenário inicial com 10%; (b) Cenário com 90% completo.

A Figura 3.13(a) mostra um cenário inicial, no qual 10% dos nós foram posicionados, o que torna simples a alocação do nó *b* em uma célula próxima ao nó *a*. Em contraste, quando a ocupação atinge 90%, conforme mostrado na Figura 3.13(b), a identificação de LUTs livres torna-se substancialmente mais difícil, em razão da elevada densidade da matriz.

Com o objetivo de verificar esse comportamento, avaliou-se o número de tentativas necessárias para o posicionamento de cada nó em dez intervalos incrementais de ocupação, interpretados como instantâneos (*snapshots*) do processo. A Figura 3.14 mostra dez *boxplots* correspondentes a esses estágios de posicionamento, desde 10% até 100% de ocupação. Observou-se que o número médio de tentativas por nó durante a busca em largura é igual a 8, enquanto o valor mediano permanece em 3. Essa discrepância decorre da presença de *outliers* associados a nós que, devido ao elevado *fan-in* e *fan-out* do grafo, necessitam alocar vizinhos em regiões já densamente ocupadas. Mesmo no estágio inicial, com apenas 10% de ocupação, observa-se a ocorrência de *outliers* que necessitam de até 100 tentativas.

Após o posicionamento de aproximadamente 50% dos nós, o comportamento deteriora-se de forma acentuada. Embora o valor mediano permaneça em 2 tentativas, o valor médio cresce para 20. Esse aumento decorre do crescimento exponencial do número de células verificadas durante a busca em largura. Considerando a métrica

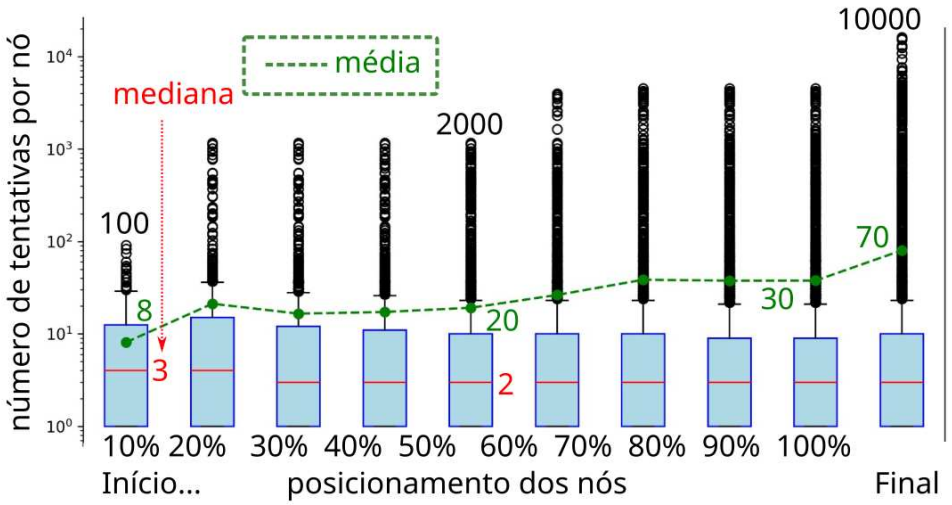


Figura 3.14: Número de tentativas com posicionamento incremental por ocupação.

de distância *Manhattan*, existem quatro LUTs a serem avaliados à distância 1, oito à distância 2, totalizando 12 verificações até esse ponto. Em distâncias maiores, como 4 ou 5, o número de células candidatas pode alcançar aproximadamente 40 ou 60, respectivamente. A quantidade máxima de buscas em relação à profundidade máxima atingida segue a seguinte equação:

$$A_{(d)} = 2d(d+1),$$

onde A_d representa a quantidade máxima de buscas acumulada da profundidade 1 a d.

A partir desse comportamento, propôs-se uma primeira estratégia de aprimoramento que procura, ao invés de realizar uma busca em largura irrestrita, limitar a busca a um limiar de distância igual a 2, sendo complementada por uma busca em profundidade direcionada, conforme ilustrado na Figura 3.15(b). A direção da busca é definida por uma heurística baseada em setores. A matriz de LUTs é particionada em 16 regiões, conforme apresentado na Figura 3.15(c). Considerando uma matriz de 32×32 LUTs e uma célula corrente localizada em $i, j = [14, 12]$, representada pelo endereço binário $[\mathbf{01}110, \mathbf{01}100]$, identifica-se o setor **01..,01..**. A busca direciona-se aos setores adjacentes com menor taxa de ocupação, informação obtida por meio de contadores simples indexados pelos dois bits mais significativos das coordenadas de linha e coluna. Nesse cenário, torna-se possível localizar uma célula livre à distância 5 em aproximadamente 15 ciclos de *clock*, em contraste com os 40 a 60 ciclos requeridos pela busca em largura tradicional.

Adicionalmente, propõe-se uma segunda estratégia para cenários de ocupação extrema, quando a taxa ultrapassa 80–90%. Conforme observado na Figura 3.14, o

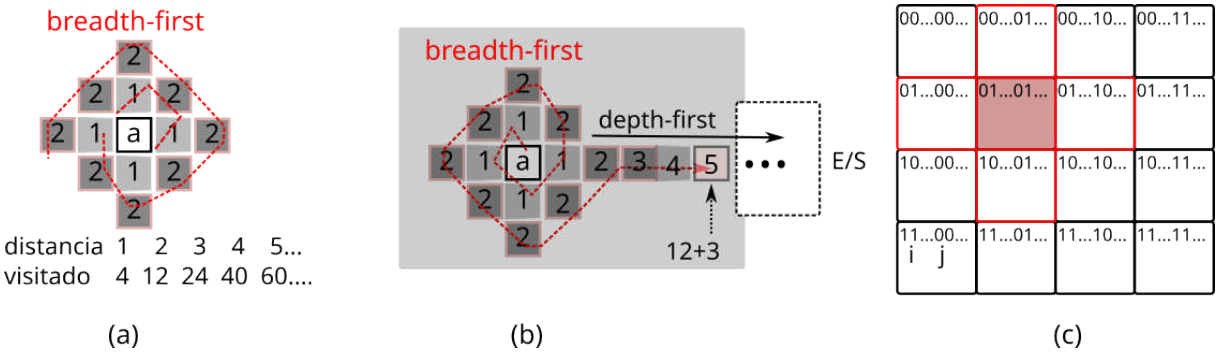


Figura 3.15: (a) Distância com BFS e distância 2; (b) Estratégia híbrida; (c) Setores.

valor médio de tentativas pode atingir 70 nesses estágios finais. O principal desafio decorre do fato de que, em matrizes FPGA altamente densas, o posicionamento incremental enfrenta dificuldades severas para localizar LUTs livres. Como alternativa, propôs-se a realização de uma varredura completa da matriz quando esse limiar de ocupação é atingido.

Em uma matriz de 32×32, essa varredura requer 1024 ciclos de *clock*. Contudo, como o mesmo número de LUTs é posicionado ao longo do processo, o custo adicional corresponde a um acréscimo médio de apenas uma tentativa por nó. As posições livres identificadas são armazenadas por quadrante em uma memória local correspondente a aproximadamente 10% do tamanho total da matriz. A partir desse ponto, as etapas subsequentes de posicionamento passam a apresentar complexidade $O(1)$, uma vez que basta selecionar uma LUT livre previamente registrada no quadrante correspondente. Dessa forma, embora o custo médio inicial aumente de maneira marginal, os nós posicionados nas etapas finais requerem apenas uma tentativa, mesmo em cenários de ocupação extrema. Os resultados destes experimentos podem ser observados na Seção 4.2.

3.3.6 *Pipeline* para os algoritmos YOTO e YOTT em QCA

Para o QCA, as arquiteturas para YOTO e YOTT reutilizam a estrutura de *pipeline* desenvolvida para CGRAs, com adaptações orientadas às restrições impostas pelos esquemas de *clocking*, em especial USE e *2DDWave*. Diferentemente do caso FPGA, as adaptações concentram-se menos no armazenamento e mais nas limitações topológicas, na direcionalidade das conexões e na propagação do sinal.

No YOTO, cada *thread* percorre um grafo de instruções definido por uma lista de arestas ordenada por uma estratégia como DFS. Em cada ciclo, identificam-se os nós a e b da aresta corrente, em que a já se encontra alocado e b representa o próximo nó a ser posicionado. O *pipeline* localiza a célula C_a associada a a e seleciona uma célula candidata C_b que seja adjacente a C_a e compatível com a orientação exigida pela célula.

A busca restringe-se às posições adjacentes de entrada ou de saída de C_a, conforme ilustrado na Figura 3.16. A seleção alterna as possibilidades por meio da função `XOR` entre o identificador da *thread* e o ponteiro de aresta.

A principal simplificação ocorre no estágio correspondente à estrutura denominada "Distância Randômica". Nos domínios CGRA e FPGA, essa estrutura assume uma memória com *offsets* que representam deslocamentos relativos válidos para derivar candidatos C_b a partir de C_a. Em QCA, a regularidade do *clocking* e a direcionalidade fixa das células eliminam essa necessidade. O estágio é implementado como um circuito combinacional que, a partir das coordenadas de C_a, indica quais posições adjacentes configuram portas válidas de entrada e de saída.

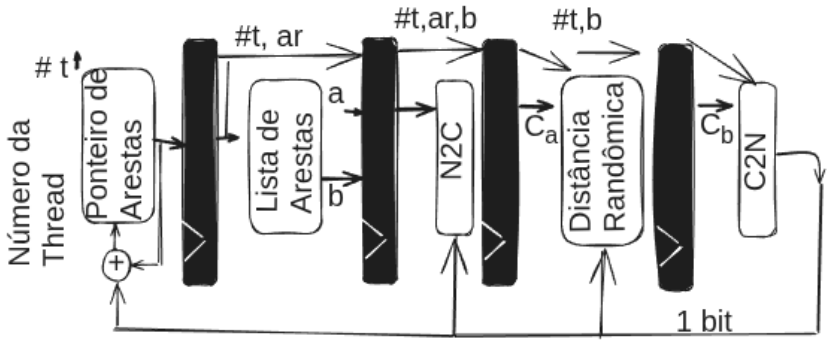


Figura 3.16: YOTO em *pipeline* no domínio QCA: busca restrita a adjacências compatíveis com orientação.

No QCA, cada célula admite, no máximo, duas entradas e duas saídas, o que permite estruturas de largura fixa para memórias como *Viz*, N_2C e C_2N. Para circuitos de maior porte, prevê-se a introdução de acesso por *cache* para manter escalabilidade, de modo análogo ao discutido no domínio FPGA.

O YOTT estende a lógica do YOTO ao incorporar restrições de distância derivadas de anotações por aresta. Cada aresta pode conter até três anotações que impõem requisitos de posicionamento relativo entre o nó atual b e até três nós previamente posicionados. As anotações são convertidas em coordenadas de referência C_1, C_2 e C_3 e propagadas ao longo do *pipeline* para verificação.

No estágio de validação, verifica-se se a célula candidata C_b atende às restrições impostas pelas anotações da aresta em processamento. Caso C_b não satisfaça os critérios, a *thread* prossegue com a busca pseudoaleatória por uma alternativa compatível. Ao final do processo, se nenhuma célula válida for identificada, seleciona-se a melhor célula candidata. Na ausência de células disponíveis, o posicionamento associado à *thread* é encerrado e caracteriza uma falha local de mapeamento. Assim como no YOTO, a estrutura "Distância Randômica" mantém-se simplificada, pois as possibilidades de alocação derivam diretamente da posição e da orientação de C_a.

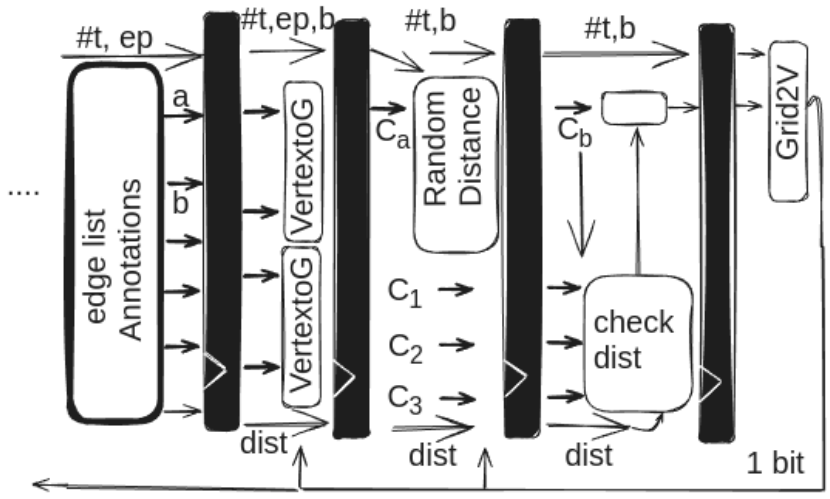


Figura 3.17: Estágio de anotações no YOTT em *pipeline* no domínio QCA.

Capítulo 4

Experimentos e Resultados

Este capítulo apresenta os resultados experimentais dos três algoritmos em três domínios de arquiteturas distintas: CGRAs,FPGAs e QCA. O capítulo é estruturado de acordo com a sequência de desenvolvimento da pesquisa. Inicialmente, na Seção 4.1, são descritos os experimentos e os resultados obtidos com a implementação do algoritmo SA para o posicionamento de um grafo de fluxo de dados a nível de instruções (DFGs) em arquiteturas CGRAs. Após essa primeira etapa, na Seção 4.1.5, são analisadas e comparadas às duas abordagens de posicionamento gulosas YOTO e YOTT.

A unidade aceleradora foi descrita no nível RTL. Adicionalmente, são conduzidos experimentos com o acelerador em *hardware* descrito em alto nível, gerados por meio da ferramenta Vitis HLS, com o objetivo de verificar as otimizações automáticas entregues por ferramentas de síntese de alto nível em comparação às obtidas por meio de descrições manuais de *hardware* em RTL.

De forma complementar, é realizada uma análise da vazão de dados resultante dos diferentes posicionamentos, com o propósito de investigar se soluções geometricamente subótimas podem proporcionar ganhos na execução de DFGs em arquiteturas CGRAs.

Na etapa seguinte, são apresentados os experimentos e resultados para o problema de posicionamento de grafos de circuitos digitais em FPGAs na Seção 4.2. Inicialmente, os resultados são comparados com a ferramenta VPR em sua versão 5 (atualmente na versão 9). Ao longo dos experimentos, são investigadas estratégias adicionais com o objetivo de viabilizar implementações mais eficientes em FPGAs, como o uso de mecanismos de memória *cache* para a redução da demanda por recursos de *hardware*. São discutidos os procedimentos adotados para a execução dos experimentos e para a comparação dos resultados obtidos.

Ainda durante a apresentação dos resultados para o posicionamento em FPGAs, são realizados novos experimentos com a versão 9 da ferramenta VPR. Os resultados obtidos demonstram que a versão mais recente apresenta melhorias significativas em termos de otimização e tempo de execução. Além disso, são avaliadas as estratégias adicionais propostas na Seção 3.3.5.

Por fim, são apresentados na Seção 4.3 os experimentos preliminares realizados no

contexto de arquiteturas QCA. O objetivo desta etapa é mostrar que o acelerador é flexível para diversas arquiteturas, apresentando a formulação e validação de funções de custo adequadas para a aplicação do algoritmo SA ao posicionamento de grafos de circuitos digitais em matrizes QCA, considerando o esquema de *clock* USE.

...

4.1 *Simulated Annealing* como Posicionador em CGRAs

Parte deste trabalho encontra-se publicada em [67], e os resultados correspondentes são apresentados nas seções subsequentes, organizadas da seguinte forma. Nas Seções 4.1.1 e 4.1.2, são descritas a metodologia adotada e os procedimentos empregados no desenvolvimento e implementação do algoritmo proposto. Na Seção 4.1.3, é apresentada a análise do consumo de recursos lógicos decorrente da implementação do acelerador em um dispositivo FPGA. Por fim, a Seção 4.1.4 apresenta uma avaliação comparativa de desempenho entre a solução desenvolvida, o software VPR [11] e uma versão paralela do algoritmo SA executada em múltiplos núcleos de CPU, conforme proposto em [17].

4.1.1 Simulador SA em *Python*

Com o objetivo de obter flexibilidade no processo de desenvolvimento, foi implementado um simulador em *Python*, destinado à avaliação e à validação de diferentes configurações do algoritmo SA em *hardware*. Esse simulador permite a experimentação de múltiplas estratégias de troca e de variações no número de estágios do *pipeline*, o que possibilita uma análise detalhada da execução. Adicionalmente, o simulador disponibiliza recursos de depuração, incluindo a visualização do conteúdo das memórias internas, o monitoramento da evolução do custo de posicionamento por *thread* e a inspeção de posicionamentos parciais ao longo da execução.

O simulador mostrou-se também essencial para a validação do gerador de código descrito na Seção 4.1.2, uma vez que viabiliza a verificação do comportamento esperado do circuito antes da síntese em *hardware*. Abordagens semelhantes são relatadas na literatura, como em [97], no qual é descrito o desenvolvimento de um simulador voltado à análise e à depuração de circuitos baseados em arranjos sistólicos.

4.1.2 Gerador de código Parametrizado

A geração do circuito foi automatizada por meio de um gerador de código implementado em *Python*, que utiliza a biblioteca Veriloggen [73]. Essa biblioteca é destinada à criação de código Verilog que, a partir de construções de alto nível em *Python*,

fornece abstrações que simplificam o desenvolvimento de projetos RTL modulares e reutilizáveis.

O gerador desenvolvido é parametrizável e permite a especificação de características do projeto, como o tamanho da arquitetura alvo, a profundidade do *pipeline*, o grau de complexidade de cada estágio e a estratégia de distribuição das memórias ao longo da arquitetura. Essa abordagem viabiliza a prototipagem de múltiplas configurações e favorece a exploração de alternativas arquiteturais, além de possibilitar a avaliação de desempenho e a utilização de recursos em diferentes cenários de implementação.

4.1.3 Recursos utilizados pela Unidade Aceleradora em *Hardware*

A síntese do código Verilog gerado para a implementação da unidade aceleradora foi realizada com o auxílio da ferramenta Vivado 2020.2, de modo a viabilizar sua implementação no FPGA Xilinx VU9P. Esse dispositivo possui aproximadamente 2,5 milhões de LUTs, 6.800 blocos DSP e 2.160 blocos de memória B-RAM, o que resulta em uma capacidade total de armazenamento em torno de 7,9 MB [99]. A Tabela 4.1 apresenta o consumo de recursos associado à implementação da unidade de posicionamento para diferentes dimensões de arquiteturas CGRA.

Tabela 4.1: FPGA Xilinx VU9P e utilização de recursos.

Tam	LUT(%)	REG(%)	Lut as RAM	Bram(%)	MEM Kb
4x4	1374 (0.1)	1574(0,1)	280 (0,1)	0 (0,0)	4
10 unidades	13289 (1.5)	15176(0,8)	2800 (0,5)	0 (0,0)	45
8x8	2120 (0.2)	1742(0,1)	784 (0,1)	0 (0,0)	13
10 unidades	20749 (2.3)	16856(0,9)	7840 (1,4)	0 (0,0)	125
9x9	2259 (0.2)	1698(0,1)	564 (0,1)	6 (0,3)	201
10 unidades	21919 (2.5)	16416(0,8)	5640 (1,0)	60 (3,2)	2010
10x10	2253 (0.2)	1698(0,1)	564 (0,1)	6 (0,3)	201
10 unidades	21879 (2.5)	16416(0,8)	5640 (1,0)	60 (3,2)	2010
12x12	3272 (0.3)	1752(0,1)	1240 (0,2)	6 (0,3)	212
10 unidades	32204 (3.6)	16956(0,9)	12400 (2,2)	60 (3,2)	2118

O consumo de recursos está diretamente associado tanto às dimensões da matriz do CGRA quanto à quantidade de unidades aceleradoras de posicionamento implementadas. Foram avaliadas arquiteturas com malhas de 4×4, 8×8, 9×9, 10×10 e 12×12 elementos de processamento, capazes de realizar o mapeamento de grafos contendo até 16, 64, 81, 100 e 144 PEs, respectivamente. O aumento da dimensão da matriz acarreta, de forma aproximadamente proporcional, a necessidade de ampliação das

memórias utilizadas para o armazenamento de dados parciais ao longo do processo de posicionamento.

Com o objetivo de avaliar a escalabilidade da proposta, também foram implementadas dez instâncias paralelas da unidade de posicionamento. Essa estratégia proporciona um aumento quase linear no desempenho, acompanhado por um crescimento igualmente linear no consumo de recursos.

A última coluna da Tabela 4.1 apresenta os valores de memória alocada para cada instância da arquitetura, permitindo avaliar a escalabilidade e o custo da proposta em termos de consumo de RAM. A demanda de memória é proporcional tanto ao número de *threads* executadas em paralelo quanto ao tamanho da matriz de posicionamento. O tamanho da matriz foi definido como potência de dois com o objetivo de explorar a codificação binária utilizada na indexação das células. Como cada *thread* possui um identificador próprio concatenado ao índice da célula, o espaço de memória cresce proporcionalmente ao número de *threads* e à dimensão da matriz. A utilização de múltiplas *threads* é necessária para ocultar a latência do *pipeline*, de modo que, após o seu preenchimento, seja possível obter uma decisão a cada passagem de uma *thread*. Embora a execução com apenas uma *thread* reduza o consumo de memória, essa configuração não permite a ocultação da latência do *pipeline*. Como consequência, os valores apresentados na tabela organizam-se em três faixas distintas de consumo, que crescem em potências de dois e correspondem às dimensões da matriz: até 4×4, até 8×8 e entre 9×9 e 15×15, conforme observado nos resultados.

O processo de síntese pode realizar a alocação das memórias tanto em LUTs quanto em blocos de B-RAM, em função do volume de armazenamento requerido. Para memórias de maior capacidade, a alocação ocorre preferencialmente em blocos de B-RAM. Entretanto, como esses blocos possuem granularidade fixa de 32 KB, pode ocorrer subutilização de recursos nos casos em que a demanda de memória seja inferior a esse valor. Por exemplo, quando são requeridos 26 KB, os 6 KB remanescentes permanecem indisponíveis para outros propósitos, caracterizando fragmentação interna. Apesar dessa potencial ineficiência, o impacto sobre a utilização global de B-RAMs permanece reduzido. Mesmo ao considerar múltiplas instâncias da unidade de posicionamento, a implementação utiliza cerca de 2,2% dos blocos disponíveis, conforme indicado na Tabela 4.1.

O principal recurso demandado corresponde ao número de LUTs, em razão da implementação da lógica de cálculo das distâncias entre os nós por meio de circuitos combinacionais. Ainda assim, mesmo no cenário mais exigente, no qual dez unidades de posicionamento operam em paralelo, o consumo de LUTs representa 3,6% da capacidade total do dispositivo.

4.1.4 Avaliação do Desempenho do Simulated Annealing (SA)

A avaliação de desempenho foi realizada em comparação com a ferramenta VPR [64], que representa o estado da arte na aplicação do algoritmo de SA ao problema de posicionamento. Essa ferramenta é atualmente empregada em uma colaboração entre o Google e a Universidade de Toronto no contexto do projeto *Antmicro*, voltado ao desenvolvimento de ferramentas de código aberto para síntese e implementação de circuitos integrados e sistemas baseados em FPGAs.

No presente estudo, as unidades aceleradoras de posicionamento em *hardware* foram sintetizadas, mapeadas e executadas em um dispositivo FPGA, com a utilização de seis *threads* em execução paralela. Os resultados experimentais indicam que cada iteração completa do algoritmo de SA é realizada em dois ciclos de *clock*, em conformidade com o comportamento previsto pela arquitetura proposta.

A Tabela 4.2 apresenta uma comparação entre os tempos de execução e a qualidade das soluções obtidas por três implementações do algoritmo SA: (i) a versão paralela em *software* com 16 *threads*, descrita em [17]; (ii) a implementação disponível na ferramenta VPR, configurada com a opção *fast* ativada[64]; e (iii) a implementação em *hardware* proposta neste trabalho. A comparação foi realizada a partir de um conjunto de grafos de fluxo de dados extraídos de aplicações reais [16], considerando 1.000 execuções do algoritmo, equivalentes a 1.000 *threads* paralelas, de modo a poder avaliar tanto a qualidade das soluções quanto o desempenho computacional.

As duas primeiras colunas da Tabela 4.2 apresentam, respectivamente, o nome da aplicação e o número de vértices do grafo correspondente. A terceira coluna indica o número de iterações executadas ao longo das 1.000 execuções. A adoção de múltiplas instâncias contribui para o aprimoramento da qualidade das soluções, em função da maior exploração do espaço de busca. A coluna $T_{16threads}$ reporta os tempos de execução totais, o tempo médio por iteração e a qualidade das soluções obtidas , tendo como métrica o tamanho máximo das filas de balanceamento, pela versão paralela em *software*, executada em um processador AMD Ryzen 7 3700X (4,4 GHz, 40 MB de *cache* L3 e 64 GB de memória RAM). Em seguida, a coluna VPR apresenta os tempos observados com a ferramenta VPR, configurada com 1.000 execuções e com a opção *fast* habilitada, bem como a qualidade das soluções em termos da profundidade das filas. Uma solução ótima não requer filas, portanto, quanto mais baixo o valor, melhor a solução.

Os resultados indicam que a versão paralela em *software* [17] apresenta, em média, um desempenho no tempo de execução 4,3 vezes superior à da ferramenta VPR configurada com a opção *fast*, além de produzir soluções de qualidade superior. Por sua vez, a implementação em *hardware* proposta neste trabalho preserva a qualidade das soluções obtidas pela versão paralela em *software*, ao mesmo tempo em que

Tabela 4.2: Comparação do Tempo de posicionamento do SA_{hard} com 2 soluções em *software*: (a) SA com 16 *threads*; (b) VPR opção *fast*. Foi considerada a execução de 1000 instâncias e seleção da melhor solução.

Bench	Nós	Iterações (milhões)	$T_{16threads}$ em *software* T(s)	T_{it} (ns)	Fila	VPR T(s)	Fila	Speedup do SA_{hard} com SA_{soft}
mac	11	11,0	0,5	47,4	0	2,0	1	5,9
simple	14	11,9	0,7	55,9	1	3,3	2	7,0
horner_bs	17	17,3	0,8	47,2	1	3,6	2	5,9
mults1	24	19,4	1,2	62,1	3	4,4	6	7,8
arf	28	26,8	1,4	53,7	3	5,2	6	6,7
conv3	28	26,8	1,4	53,8	1	5,3	4	6,7
motion_vec	32	27,9	1,5	53,1	0	5,5	1	6,6
fir2	40	37,4	2,0	53,7	3	6,8	4	6,7
fir1	44	38,2	2,1	55,8	1	6,8	3	7,0
fdback_pts	54	49,6	2,6	52,1	4	8,9	7	6,5
k4n4op	59	50,5	3,3	65,2	4	11,2	6	8,1
h2v2_smo	62	50,8	3,1	60,6	3	10,4	7	7,6
cosine1	66	61,7	3,6	57,9	2	13,3	5	7,2
ewf	66	61,8	3,7	59,3	4	14,0	8	7,4
Cplx8	77	64,3	4,1	64,2	3	16,0	6	8,0
Fir16	77	64,3	4,2	64,8	5	16,8	8	8,1
cosine2	81	64,5	4,2	65,4	4	15,6	7	8,2
FilterRGB	84	77,8	4,6	58,9	6	21,7	10	7,4
Collapse	105	95,2	5,8	60,5	5	23,7	7	7,6
imperpolate	108	96,0	5,3	55,5	2	21,4	4	6,9
w_bmp_head	110	96,5	5,0	52,0	5	26,9	10	6,5
matmul	116	97,1	6,0	61,8	4	26,1	9	7,7
fir16_collapse	182	157,6	10,3	65,1	6	49,5	10	8,1
cplx8_interpolate	185	157,5	9,8	62,1	7	49,6	13	7,8
smooth_color_z	196	158,6	10,0	62,8	4	51,5	10	7,9
jpg_fdct_islow	217	181,1	12,2	67,4	8	67,9	14	8,4
idctcol	298	259,1	16,5	63,9	11	95,3	19	8,0
jpeg_idct_ifast	303	260,1	16,6	63,7	11	81,0	20	8,0
invert_matrix	357	292,2	18,7	64,0	8	93,4	12	8,0
Cplx8x8	616	506,8	35,1	69,3	5	192,5	12	8,7
Fir16x8	616	506,8	35,1	69,3	4	201,7	11	8,7
AVG	137,8	117,0	7,5	59,6	4,1	37,1	7,9	7,5
GEO	82,4	72,1	4,3	59,3	3,6	18,1	6,3	7,4

proporciona ganhos adicionais de desempenho, detalhados a seguir.

A síntese da arquitetura mostra que o circuito da unidade aceleradora opera com uma frequência mínima de 250 MHz, o que viabiliza a execução de uma iteração completa do algoritmo de SA, com ou sem troca, em 8 nanosegundos. Em contraste, o tempo médio por iteração da versão em *software* com 16 *threads* é de 59,3 nanosegundos, o que corresponde a um ganho de desempenho de aproximadamente 7,4 vezes. Quando comparada à ferramenta VPR, a implementação em *hardware* alcança

um ganho médio geométrico de desempenho de 31,5 vezes.

Adicionalmente, a replicação da unidade de SA no FPGA possibilita maior escalabilidade de desempenho. Quando são empregadas dez unidades com operação em paralelo, o sistema alcança um aumento de velocidade de até 315 vezes em relação à ferramenta VPR, mantendo ou superando a qualidade das soluções obtidas. Esses resultados mostram a viabilidade e a eficiência da proposta de implementação do algoritmo de SA em *hardware* para a resolução de problemas de posicionamento em arquiteturas reconfiguráveis.

4.1.5 YOTO e YOTT para CGRAs como Posicionadores para CGRAs

Como continuidade dos experimentos apresentados na Seção 4.1, esta seção apresenta e discute os resultados obtidos a partir das implementações de unidades aceleradoras em hardware para os algoritmos YOTO e YOTT, cujos desempenhos são analisados em comparação àqueles previamente observados para a implementação do algoritmo SA.

As unidades aceleradoras propostas para os algoritmos YOTO e YOTT estão implementadas, com os respectivos resultados experimentais organizados e prontos para futura publicação. Nesta seção, descrevem-se os experimentos realizados e apresentam-se os resultados obtidos.

Na Seção 4.1.6 são descritos os *benchmarks* utilizados nos experimentos, os quais englobam instâncias extraídas da literatura e instâncias sintéticas, elaboradas de modo a representar diferentes perfis estruturais de grafos. Nessa seção, também são discutidos os critérios adotados para a seleção desses *benchmarks*.

A Seção 4.1.7 apresenta os resultados dos mapeamentos obtidos pelos algoritmos SA, YOTO e YOTT, com ênfase na análise da qualidade das soluções geradas.

Na sequência, a Seção 4.1.8 detalha a utilização de recursos lógicos e de memória decorrentes das implementações da unidade aceleradora em *hardware* dos algoritmos, com o objetivo de avaliar a viabilidade e a escalabilidade das soluções propostas em dispositivos FPGAs.

A Seção 4.1.9 apresenta uma análise comparativa do desempenho das implementações dos algoritmos em *software* e em *hardware*, considerando o tempo de execução, o consumo de recursos e a qualidade das soluções obtidas. Por sua vez, a Seção 4.1.10 estabelece uma comparação direta entre as soluções desenvolvidas e versões equivalentes sintetizadas com a ferramenta Vitis HLS, o que permite a análise das diferenças em termos de custo de síntese e desempenho do projeto.

Por fim, a Seção 4.1.11 discute os resultados sob a perspectiva da vazão de dados dos DFGs, com o objetivo de avaliar como os algoritmos de posicionamento impactam o paralelismo e o desempenho global do sistema reconfigurável.

4.1.6 *Benchmarks*

Os aceleradores propostos para os algoritmos YOTO e YOTT, descritos nas Seções 3.3.1 e 3.3.2, respectivamente, foram avaliados e comparados à unidade de posicionamento baseada no algoritmo SA, apresentada na Seção 3.2.1. Para essa avaliação, foi empregada uma coleção de grafos extraídos da literatura [5, 15].

A Tabela 4.3 apresenta as principais características de cada *benchmark*, como o número de nós, o número total de arestas, a quantidade de arestas não visitadas pelos algoritmos de travessia e o número de portas de entrada e saída. No caso do algoritmo YOTO, nem todas as arestas são visitadas, uma vez que o algoritmo realiza apenas uma única visita a cada vértice do grafo, o que pode resultar na não consideração de parte das arestas durante a etapa de posicionamento.

Tabela 4.3: Conjunto de *benchmarks*.

Bench	Entradas	Saídas	nós	Arestas	Não visitadas
mac	3	1	11	11	1
simple	4	1	16	17	2
horner_bs	4	1	17	16	0
mults1	1	1	24	27	4
arf	8	2	28	30	3
conv3	9	1	30	32	3
fir2	16	1	40	39	0
fir1	22	1	44	43	0
k4n4op	22	2	59	74	16
h2v2_smooth	16	1	62	65	4
ewf	2	5	69	82	14
Fir16	1	1	77	91	15
Cplx8	1	1	77	91	15
cosine2	31	8	83	93	11
FilterRGB	2	1	84	97	14
matmul	24	4	116	124	9
jpeg_fdct_islow	24	8	217	254	38
idctcol	18	8	298	348	51
jpeg_idct_ifast	24	8	303	346	44
invert_matrix	76	16	368	390	23

Como a exploração exaustiva do espaço de soluções se torna inviável na prática para grafos com dezenas de nós (20 ou mais nós), adotou-se a estratégia proposta em [19], fundamentada na construção de *benchmarks* sintéticos cujas soluções ótimas são previamente conhecidas. A geração desses grafos segue um procedimento inverso ao convencional: parte-se de um posicionamento válido em uma matriz de células, onde a priori existe uma solução e sabe-se o seu custo, e a partir dessa solução deriva-se o grafo correspondente.

Inicialmente, um nó é alocado de forma aleatória em uma célula da matriz. Na sequência, novos nós são inseridos apenas quando podem ser posicionados em células adjacentes às já ocupadas, o que assegura a formação de uma estrutura localmente coerente. A Figura 4.1 (a) ilustra um exemplo de grafo ideal gerado por esse processo, caracterizado pela ocupação de uma área mínima na matriz.

Apesar de serem construídos a partir de posicionamentos ótimos, os grafos sintéticos não garantem que a solução correspondente seja prontamente identificada por algoritmos de travessia. Isso ocorre porque a noção de localidade implícita ao posicionamento original pode ser parcialmente perdida. Por exemplo, um nó inicialmente alocado de forma adjacente a outro pode ser reposicionado em qualquer uma das células vizinhas, como leste ou oeste, sem que haja uma indicação explícita de qual dessas alternativas preserva, de forma mais adequada, o posicionamento ótimo original.

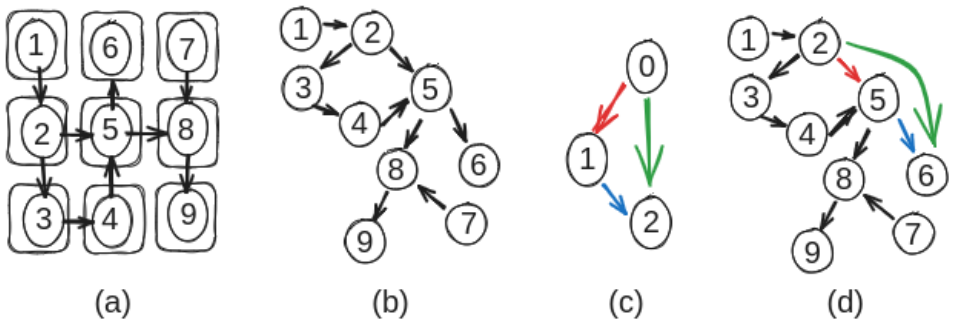


Figura 4.1: (a) Exemplo de grafo ideal para a matriz; (b) Grafo extraído que possui posicionamento ótimo na matriz; (c) Exemplo de grafo sem solução ideal na matriz.

A Tabela 4.4 apresenta as propriedades estruturais dos grafos sintéticos utilizados nos experimentos. Os *benchmarks* identificados pelo prefixo $No_{V_X E_Y}$ correspondem a grafos para os quais não existe um posicionamento ótimo, construídos intencionalmente com essa finalidade. Por sua vez, os *benchmarks* identificados como $Opt_{V_X E_Y}$ possuem um posicionamento ótimo conhecido. Em ambas as nomenclaturas, os índices V_X e E_Y indicam, respectivamente, o número de vértices e o número de arestas do grafo. Adicionalmente, foram incluídos *benchmarks* que apresentam padrões topológicos nos quais não existe solução viável sem a utilização de filas. A coluna *Não Visitadas* indica a quantidade de arestas que não são percorridas pelo algoritmo YOTO, uma vez que esse algoritmo visita cada vértice apenas uma única vez e a ordem de travessia, bem como o subconjunto de arestas selecionadas, depende de decisões aleatórias.

Qualquer grafo que contenha o subgrafo com um clique de três nós, ilustrado na Figura 4.1 (c) para a rede de conexão em malha irá requerer o uso de filas. A Figura 4.1 (d), mostra um exemplo de um grafo que possui uma clique com os vértices 2,5 e 6, que requer no mínimo uma fila para equilibrar os caminhos.

Tabela 4.4: Conjunto de grafos gerados otimizados.

Bench	Entradas	Saídas	Nós	Arestas	Não Visitadas
$No_{V_{14}E_{16}}$	3	3	14	16	3
$Opt_{V_{15}E_{14}}$	1	6	15	14	0
$Opt_{V_{15}E_{17}}$	3	4	15	17	3
$No_{V_{18}E_{18}}$	1	6	18	18	1
$No_{V_{20}E_{20}}$	1	7	20	20	1
$No_{V_{22}E_{25}}$	2	7	22	25	4
$Opt_{V_{24}E_{28}}$	2	4	24	28	5
$Opt_{V_{25}E_{24}}$	1	11	25	24	0
$No_{V_{30}E_{33}}$	1	7	30	33	4
$No_{V_{33}E_{33}}$	1	11	33	33	1
$Opt_{V_{34}E_{38}}$	3	8	34	38	5
$Opt_{V_{40}E_{39}}$	1	11	40	39	0
$No_{V_{42}E_{42}}$	1	13	42	42	1
$No_{V_{43}E_{50}}$	4	10	43	50	8
$Opt_{V_{50}E_{49}}$	1	16	50	49	0
$Opt_{V_{59}E_{68}}$	4	13	59	68	10

Outros padrões estruturais incluem árvores com grau de vértices (15 ou mais), comumente observadas em grafos oriundos de aplicações de processamento de dados. Para esse tipo de grafo, a obtenção de um posicionamento ótimo, no sentido de minimizar a soma das distâncias entre vértices adjacentes dentro de uma área retangular compacta, constitui um problema complexo [70]. A inclusão desses casos tem como objetivo avaliar a robustez dos algoritmos em relação à capacidade de generalização e à qualidade das soluções produzidas em cenários que se aproximam de aplicações reais.

4.1.7 Resultados de Mapeamento

A Tabela 4.5 apresenta os resultados do mapeamento dos grafos de fluxo de dados obtidos pelos três algoritmos de posicionamento implementados. A qualidade das soluções foi avaliada com base no comprimento total das conexões, definida como a soma dos custos de todas as arestas mapeadas na matriz, subtraída do número total de arestas. Uma solução ótima corresponde ao valor zero (0), o que indica que todas as arestas conectam nós alocados em células adjacentes.

Na tabela, o valor do comprimento total, que é a soma de todas as conexões, é acompanhado, entre parênteses, pelo número de arestas não posicionadas em células adjacentes. Por exemplo, o valor “12 (7)” indica que, para o *benchmark mac*, o algoritmo SA obteve um comprimento total de 12 segmentos adicionais, além do número total de arestas, onde sete arestas foram mapeadas com mais de um segmento. Por exemplo, se

a aresta a necessitou de 4 segmentos de conexão adicionais, a aresta b usou 3 segmentos adicionais e as arestas c, d, e, f, e g usaram um segmento adicional, totalizando sete arestas e 12 segmentos. Na última coluna da tabela é indicada a arquitetura empregada, sendo "M" referente à malha ou grade (*mesh or grid*) e "H" referente à arquitetura *one-hop*. Vale ressaltar que uma clique de três nós requer uma fila para a arquitetura em malha, mas pode ter solução sem fila na arquitetura *one-hop*.

Tabela 4.5: Comprimento total e Número de arestas roteadas para os algoritmos SA, YOTO e YOTT implementados em *hardware pipeline*.

	SA				YOTO			YOTT			
Bench	**1**	**6**	**60**	**600**	**6**	**60**	**600**	**10**	**100**	**1000**	
mac	0 (0) 0 (0)	0 (0) 0 (0)	0 (0) 0 (0)	0 (0) 0 (0)	2 (2) 1 (1)	0 (0) 0 (0)	0 (0) 0 (0)	2 (1) 0 (0)	0 (0) 0 (0)	0 (0) 0 (0)	M H
$Opt_{V_{15}E_{14}}$	0 (0) 0 (0)	0 (0) 0 (0)	0 (0) 0 (0)	0 (0) 0 (0)	2 (2) 0 (0)	0 (0) 0 (0)	0 (0) 0 (0)	0 (0) 0 (0)	0 (0) 0 (0)	0 (0) 0 (0)	M H
horner_bs	0 (0) 0 (0)	0 (0) 0 (0)	0 (0) 0 (0)	0 (0) 0 (0)	0 (0) 0 (0)	0 (0) 0 (0)	0 (0) 0 (0)	0 (0) 0 (0)	0 (0) 0 (0)	0 (0) 0 (0)	M H
$No_{V_{18}E_{18}}$	1 (1) 0 (0)	1 (1) 0 (0)	1 (1) 0 (0)	1 (1) 0 (0)	3 (3) 0 (0)	1 (1) 0 (0)	1 (1) 0 (0)	3 (2) 0 (0)	1 (1) 0 (0)	1 (1) 0 (0)	M H
$Opt_{V_{25}E_{24}}$	4 (3) 0 (0)	1 (1) 0 (0)	0 (0) 0 (0)	0 (0) 0 (0)	5 (2) 4 (4)	3 (3) 1 (1)	1 (1) 0 (0)	7 (6) 1 (1)	5 (3) 1 (1)	2 (2) 0 (0)	M H
arf	5 (5) 1 (1)	4 (4) 0 (0)	4 (4) 0 (0)	4 (3) 0 (0)	16 (9) 5 (5)	9 (6) 2 (2)	8 (6) 2 (2)	11 (7) 3 (2)	7 (6) 2 (1)	7 (4) 2 (2)	M H
conv3	3 (3) 0 (0)	4 (4) 0 (0)	3 (3) 0 (0)	3 (3) 0 (0)	14 (6) 6 (5)	11 (5) 3 (3)	10 (7) 2 (2)	15 (7) 3 (2)	9 (6) 1 (1)	5 (3) 0 (0)	M H
$No_{V_{30}E_{33}}$	5 (5) 0 (0)	2 (2) 0 (0)	1 (1) 0 (0)	1 (1) 0 (0)	12 (7) 1 (1)	4 (3) 3 (3)	3 (2) 1 (1)	9 (5) 3 (3)	6 (5) 2 (2)	4 (3) 1 (1)	M H
$No_{V_{43}E_{50}}$	6 (4) 3 (3)	1 (1) 0 (0)	1 (1) 0 (0)	1 (1) 0 (0)	21 (10) 8 (8)	20 (11) 8 (5)	14 (10) 6 (6)	25 (12) 7 (7)	17 (11) 7 (5)	15 (11) 5 (4)	M H
$Opt_{V_{59}E_{68}}$	13 (9) 2 (2)	16 (10) 2 (1)	8 (8) 0 (0)	3 (2) 0 (0)	45 (17) 19 (14)	33 (16) 15 (9)	30 (17) 13 (11)	37 (18) 19 (14)	25 (18) 13 (10)	29 (18) 11 (8)	M H
matmul	54 (29) 19 (15)	55 (28) 11 (10)	47 (29) 8 (6)	42 (27) 5 (5)	95 (28) 45 (21)	98 (26) 41 (20)	79 (25) 33 (20)	107 (30) 41 (18)	89 (31) 34 (17)	82 (23) 30 (16)	M H
invert_matrix	1697 (159) 845 (181)	1427 (145) 1198 (260)	1346 (141) 683 (107)	1235 (153) 1053 (242)	409 (90) 181 (70)	370 (87) 178 (64)	330 (88) 139 (61)	374 (94) 167 (61)	340 (95) 158 (64)	332 (89) 148 (58)	M H

A tabela encontra-se organizada em função do tamanho dos grafos de fluxo de dados (DFGs), com os grafos de menor porte dispostos na parte superior e os de maior porte na parte inferior. Para o algoritmo SA, os resultados são apresentados considerando execuções com 1, 6, 60 e 600 instâncias independentes do algoritmo. O número de trocas realizadas varia de acordo com o tamanho do grafo, sendo observado um valor médio da ordem de 10.000 trocas por nó.

Em arquiteturas do tipo *one-hop*, o algoritmo SA atinge a solução ótima para todos os *benchmarks* com menos de 60 nós, com exceção dos dois maiores grafos avaliados. Em diversos casos, uma única execução do SA mostra-se suficiente para alcançar a solução ótima. Em contraste, a obtenção de soluções ótimas em arquiteturas do tipo *mesh* apresenta maior complexidade, como evidenciado pelos *benchmarks arf* e *conv3*, que possuem 28 e 30 nós, respectivamente, e não apresentam clique. Mesmo após 600 execuções, não foi identificada, para esses casos, uma solução ótima de mapeamento

que dispense o uso de segmentos adicionais. Ressalta-se que a solução ótima pode exigir a utilização de filas; contudo, a análise realizada considera exclusivamente a minimização do número de segmentos utilizados.

Considerando que a solução ótima em termos de segmentos adicionais é um problema em aberto, foram empregados grafos sintéticos cuja solução ótima é conhecida a priori, com o objetivo de validar a capacidade do algoritmo SA. Como exemplo, o grafo $Opt_{V_{25}E_{24}}$ foi construído como explicado na seção 4.1.6 a partir da solução na arquitetura e possui pelo menos um mapeamento ótimo, composto por 25 vértices e 24 arestas. O mapeamento ótimo foi identificado após apenas 60 execuções do SA em uma arquitetura do tipo *mesh*. Em contraste, o grafo $Opt_{V_{59}E_{68}}$ não atingiu sua solução ótima mesmo após 600 execuções, mantendo duas arestas fora de suas posições ideais. Outros grafos sintéticos, como $No_{V_{18}E_{18}}$, $No_{V_{30}E_{33}}$ e $No_{V_{43}E_{50}}$, apresentaram custo mínimo igual a 1 e alcançaram suas melhores soluções com 60 execuções ou menos em arquiteturas do tipo *mesh*. Esse comportamento decorre da presença de cliques nesses grafos. Já em arquiteturas do tipo *one-hop*, onde as cliques podem ser resolvidas, os mesmos grafos atingiram posicionamento ótimo com seis execuções ou menos.

As colunas subsequentes da Tabela 4.5 apresentam os resultados obtidos pelos algoritmos YOTO e YOTT. A profundidade do *pipeline* define o número de instâncias avaliadas simultaneamente: seis para o YOTO e dez para o YOTT. Ambos os algoritmos apresentam desempenho satisfatório para grafos com menos de 20 nós, com a obtenção consistente de soluções ótimas. Para grafos de porte intermediário, entre 20 e 30 nós, o YOTO ainda produz soluções próximas do ótimo, porém com degradação progressiva à medida que o tamanho do grafo aumenta. Por exemplo, nos grafos $No_{V_{43}E_{50}}$ e $Opt_{V_{59}E_{68}}$, as melhores soluções produzidas pelo YOTO deixam de posicionar corretamente 8 e 14 arestas, respectivamente. Em contraste, o algoritmo SA atinge a solução ótima ou falha em posicionar apenas uma aresta nesses casos. Entretanto, o SA executa, em média, mais de 1.000 iterações por nó, o que resulta em um tempo de execução aproximadamente três ordens de grandeza superior ao das abordagens baseadas em travessia.

Os resultados mostram um *trade-off* entre a qualidade das soluções e o tempo de execução. Enquanto o algoritmo SA produz soluções de qualidade ao custo de um maior esforço computacional, os algoritmos baseados em travessia geram soluções próximas do ótimo com um tempo de execução reduzido, mostrando-se mais adequados para aplicações que demandam respostas rápidas ou mapeamento *on-the-fly*.

Por fim, os dois maiores *benchmarks* considerados mostram os limites práticos de ambas as abordagens. Nenhuma das estratégias, seja o SA ou os algoritmos de travessia, foi capaz de atingir o posicionamento ótimo. De forma particular, para o grafo *invert_matrix*, com 368 nós, o YOTO apresentou desempenho superior ao do SA,

o que indica maior capacidade de generalização em cenários extremos. Em contraste, para o grafo *matmul*, o algoritmo SA obteve uma solução mais próxima do ótimo na arquitetura do tipo *one-hop*, porém com um tempo de execução elevado.

4.1.8 Utilização de recursos em FPGAs dos Aceleradores de CGRA

A Tabela 4.6 apresenta os recursos necessários para a implementação dos três aceleradores SMT: SA, YOTO e YOTT; em um dispositivo FPGA de grande porte. As colunas LUTs, Regs e B-RAMs indicam, respectivamente, as quantidades e os percentuais de utilização dos recursos em termos de LUTs, registradores e blocos de memória (B-RAMs). Foram consideradas configurações com 5, 10 e 15 unidades de execução (**UN**) para cada acelerador, bem como variações no tamanho da matriz de posicionamento (**MP**), uma vez que a dimensão da arquitetura CGRA impacta diretamente o dimensionamento das memórias internas.

Tabela 4.6: Recursos utilizados para a síntese dos algoritmos de posicionamento SMT no Intel Arria 10:10AX115U3F45E2SGE3 mapeado no Quartus 16.

Pos.	UN	MP	LUT	Regs	B-RAMs
SA	5	10x10	15,132	21,701	160,480 (100xM20K)
	10	10x10	30,271	43,161	320,960 (200xM20K)
	15	10x10	46,722	64886	481,440 (300xM20K)
YOTO	5	10x10	1163	1566	148,840 (20xM20K)
	10	10x10	2,283	3,131	297,680 (60xM20K)
	15	10x10	3,306	4,696	446,520 (90xM20K)
YOTT	5	10x10	3,244	4,006	903,020 (85xM20K)
	10	10x10	6,116	8,011	1,806,040 (170xM20K)
	15	10x10	8,728	12,016	7,709,060 (255xM20K)

Os resultados evidenciam a eficiência da abordagem baseada em B-RAMs no que se refere à economia de recursos lógicos e de registradores, os quais podem, dessa forma, ser realocados para a implementação do próprio CGRA. A unidade aceleradora do algoritmo YOTO apresenta o menor consumo de recursos entre as soluções avaliadas, utilizando aproximadamente cinco vezes menos memória e mais de dez vezes menos LUTs e registradores quando comparada à unidade aceleradora do algoritmo SA. De forma análoga, a unidade aceleradora do YOTT consome cerca de dez vezes menos LUTs do que a unidade SA, embora requeira quantidade semelhante de B-RAM, em decorrência da necessidade de armazenamento de anotações e de estruturas auxiliares destinadas à verificação de restrições de posicionamento.

Outro aspecto relevante refere-se à escalabilidade das três arquiteturas, uma vez que o consumo de recursos apresenta crescimento aproximadamente linear em função

do número de unidades de execução. A maior configuração avaliada, composta por 15 unidades do acelerador SA, utiliza cerca de 10% das LUTs disponíveis no FPGA alvo. Em contraste, a configuração com cinco unidades do acelerador YOTO consome menos de 1% das LUTs, o que evidencia sua adequação para aplicações embarcadas ou para cenários caracterizados por restrições na utilização de recursos.

4.1.9 Tempo de Execução

A avaliação de desempenho dos aceleradores implementados em *hardware*, em termos do tempo de execução, foi realizada por meio de comparações com implementações em *software* dos algoritmos SA, YOTO e YOTT. A implementação em *software* do algoritmo SA, adotada como referência (*baseline*), foi executada em um processador *multicore* com suporte a 16 *threads*, seguindo o pseudocódigo apresentado na Figura 2.3 (a). Para os algoritmos baseados em travessia, as comparações foram conduzidas com base na implementação descrita em [15].

Tabela 4.7: Tempo médio de execução, número de tentativas e trocas por nó.

Instâncias do Posicionador *Soft* ou *Hard*	**Tempo Total**	**Trocas de nós**		**Número de Unidades**
		Tempo	**Tentativas**	
SA60 16-th *soft*	184ms	60ns	846	1
SA60 *hard*	24.3ms	8ns	846	1
SA60 *hard*	2.4ms	0.8ns	846	10
YOTO60 *soft*	498μs	7ns	18	1
YOTO60 *hard*	1.9μs	0.4ns	8	10
YOTT100 *soft*	1.91ms	19ns	16	1
YOTT100 *hard*	2.5μs	0.4ns	10	10

A Tabela 4.7 apresenta os tempos médios de execução obtidos nos experimentos realizados com grafos contendo aproximadamente 60 nós, os quais correspondem a instâncias cujo espaço de busca é suficientemente amplo para tornar inviável a aplicação de métodos exatos. A primeira coluna da tabela especifica o algoritmo considerado, o número de instâncias avaliadas e o tipo de implementação empregada (*hardware* ou *software*). Por exemplo, a notação *SA60 Hard* indica a execução de 60 instâncias do algoritmo SA em *hardware*.

A coluna **Tempo total** apresenta o tempo médio de execução para o conjunto de instâncias avaliadas. Como linha de base, considera-se a versão **SA60**, executada em *software* com 16 *threads* em um processador AMD Ryzen operando a 3 GHz, a qual apresenta desempenho aproximadamente três vezes superior ao da ferramenta VPR [11], tradicionalmente reconhecida como o estado da arte em posicionamento baseado em SA. A implementação do algoritmo SA em *hardware*, composta por uma

única unidade com *pipeline* de seis estágios, supera a execução em CPU com 16 *threads* por um fator de 7,57. Com a replicação da unidade de posicionamento para dez instâncias paralelas, obtém-se uma aceleração de até 75,7 vezes em relação à linha de base em *software*.

Os algoritmos YOTO e YOTT apresentam desempenho superior ao do SA, sobretudo em função da natureza gulosa e incremental de suas estratégias de resolução. O algoritmo YOTO, mesmo quando executado em *software*, alcança desempenho até duas ordens de grandeza superior ao do SA. A implementação em *hardware* amplia esse ganho e permite que seis instâncias do *pipeline* concluam a execução em menos de 450 ciclos. Dessa forma, o tempo total de execução é reduzido a poucos microssegundos para grafos com 60 a 100 nós, o que corresponde a uma diminuição de até 100 vezes em relação à versão em *software* do YOTO apresentada em [15].

A coluna **Trocas por nó** da Tabela 4.7 apresenta o número médio de tentativas de troca por nó, bem como o tempo requerido para a realização de cada tentativa. Embora a frequência de operação dos FPGAs seja inferior à de processadores convencionais (em torno de 200 MHz frente a aproximadamente 3 GHz), a implementação em *hardware* permite a execução de uma tentativa de troca em apenas dois ciclos de *clock* (8 ns). Em contraste, a versão em *software*, executada com 16 *threads*, requer cerca de 60 ns por tentativa. Dessa forma, mesmo sob a limitação de frequência de operação, a solução em *hardware* apresenta um ganho de desempenho para o tempo de execução aproximado de 7 vezes em relação à sua contraparte em *software*.

As arquiteturas baseadas em travessia (YOTO e YOTT) apresentam um tempo ainda mais reduzido, uma vez que realizam uma tentativa de posicionamento de nó por ciclo de *clock* (4 ns), o que corresponde a 0,4 ns por operação quando se empregam dez unidades paralelas. A complexidade computacional do posicionamento para os métodos de travessia é $O(N)$, em que N denota o número de nós do grafo. Em média, cada nó requer entre 8 e 16 ciclos para ser posicionado, o que representa uma redução expressiva do tempo de execução em comparação ao SA, cujo número de tentativas por nó pode variar entre 800 e 10.000.

Embora a redução do tempo de execução possa implicar uma leve degradação da qualidade do posicionamento em determinados cenários, os resultados experimentais indicam que os algoritmos baseados em travessia apresentam uma relação custo-benefício favorável entre o tempo de execução e a qualidade das soluções. Dessa forma, as implementações em *hardware* dos algoritmos YOTO e YOTT configuram-se como alternativas promissoras para aplicações que demandam mapeamento em tempo real ou que operam sob restrições rigorosas de tempo de execução.

4.1.10 Projeto da Unidade Aceleradora com Descrição em Vitis HLS

Com o objetivo de avaliar a qualidade do código RTL das implementações propostas, realizou-se uma comparação com circuitos gerados pela ferramenta Vitis HLS, da Xilinx, amplamente empregada no desenvolvimento de *hardware* a partir de descrições em linguagens de alto nível. Em contraste com a abordagem fundamentada na geração automática por meio de modelagem em *Python*, proposta nesta tese, com o uso da biblioteca Veriloggen [73], esta análise baseou-se em implementações em *C++* que utilizam estruturas de dados elementares, como vetores, para a representação das informações de posicionamento.

A Tabela 4.8 apresenta uma comparação entre as versões geradas por meio do Vitis HLS e os módulos RTL obtidos diretamente com Veriloggen, com destaque para as diferenças observadas na quantidade de recursos de *hardware* empregados e no desempenho alcançado.

Tabela 4.8: Implementações RTL e HLS: Recursos e desempenho para o FPGA Xilinx Virtex ™XCU55 UltraScale+ FPGA.

Versão	**Swap (clk)**	**LUT**	**Regs**	**B-RAM**
SA sem pragmas	28	18,498	28,947	9
SA pragmas	12	50,810	28,300	0
SA Python-RTL	2	1,953	1,538	6
YOTO Python-RTL	1	193	315	2
YOTT Python-RTL	1	552	699	7

Foram desenvolvidas duas variantes do algoritmo SA por meio da ferramenta Vitis HLS: uma versão básica, sem o emprego de diretivas de otimização (*pragmas*), e uma versão otimizada, na qual foram inseridas manualmente diretivas para controle de paralelismo, aplicação de *pipelining* e gerenciamento de latência. A implementação sem *pragmas* apresentou latência de 28 ciclos de *clock* por tentativa de troca, enquanto a versão otimizada reduziu esse valor para 12 ciclos. Apesar dessa redução, ambas as versões mostraram desempenho inferior ao da implementação em RTL desenvolvida em *Python*, a qual realiza cada troca em apenas dois ciclos. Do ponto de vista de desempenho, as implementações geradas via HLS apresentaram latências aproximadamente seis vezes superiores à implementação em RTL proposta neste trabalho.

Além do impacto observado no tempo de execução para um *swap*, constatou-se um aumento significativo no consumo de recursos lógicos. A versão otimizada do algoritmo SA, gerada por meio do HLS, apresentou uma utilização de até 26 vezes mais LUTs em comparação com a implementação em RTL. Esse comportamento decorre, em parte, da estratégia adotada pelo compilador HLS, que tende a mapear estruturas de dados, como vetores, diretamente em lógica combinacional baseada em LUTs, em

detrimento da alocação em blocos de memória B-RAM. Tal decisão de mapeamento compromete a escalabilidade da solução, sobretudo em arquiteturas que demandam múltiplas instâncias paralelas.

As três últimas linhas da Tabela 4.8 apresentam uma comparação quantitativa dos recursos de *hardware* empregados pelas implementações em RTL dos aceleradores baseados nos algoritmos SA, YOTO e YOTT. Observa-se que, para uma mesma instância, a unidade em RTL do algoritmo YOTO requer aproximadamente 200 vezes menos LUTs do que a versão otimizada em HLS do algoritmo SA, além de utilizar cerca de três vezes menos blocos de memória B-RAM do que a própria implementação em RTL do SA. Essa expressiva redução no consumo de recursos de lógica programável e de memória reforça a viabilidade técnica da geração automatizada de circuitos em RTL.

Todas as sínteses foram realizadas com o uso da ferramenta Vivado 2022.2 (64 b), da Xilinx, direcionadas ao dispositivo FPGA Xilinx UltraScale+ VU9P.

4.1.11 Vazão de dados dos Grafos mapeados no CGRA

Embora o posicionamento ideal em arquiteturas CGRA priorize, de modo geral, a minimização das distâncias entre nós, com a alocação preferencial em células adjacentes (distância unitária), a taxa de vazão (*throughput*) na execução dos DFGs não apresenta, necessariamente, uma correlação direta com o comprimento mínimo das conexões. A vazão efetiva é determinada, sobretudo, pelo balanceamento dos caminhos de dados, uma vez que reconvergências de fluxos em pontos de saída comuns podem introduzir penalidades temporais significativas.

Ao considerar, por exemplo, um cenário no qual dois caminhos convergem em um mesmo nó de saída, denotados como **A** e **B**, mesmo que o caminho A apresente exclusivamente conexões com comprimentos ideais, a existência de um caminho **B** mais longo implica a redução da taxa de vazão, uma vez que a execução associada ao caminho **A** permanece condicionada à finalização do caminho B. Em contraste, quando ambos os caminhos encontram-se balanceados, mesmo que incorporem custos de conexão não ideais, o tempo total de processamento pode ser reduzido, pois os dados passam a alcançar o nó de destino de forma aproximadamente simultânea. Dessa forma, o balanceamento dos caminhos assume relevância comparável à minimização das distâncias de conexão para a otimização da eficiência global do circuito.

A Tabela 4.9 apresenta uma análise comparativa entre o melhor rendimento obtido segundo o critério de minimização da distância das interconexões (posicionamento mais compacto) e o melhor rendimento absoluto, independentemente da otimização dessas distâncias para a arquitetura *mesh*. A comparação foi realizada com diferentes quantidades de instâncias dos algoritmos SA, YOTO e YOTT, definidas de modo a equalizar o tempo total de execução entre as abordagens.

Tabela 4.9: Avaliação de rendimento para as três posicionamento: SA com 6 *threads*, YOTO com 600 *threads* e YOTT com 100 *threads*.

Bench	Melhor Distância			Melhor vazão		
	SA6	**YOTO600**	**YOTT100**	**SA6**	**YOTO600**	**YOTT100**
mac	99.40	99.40	**99.88**	99.40	99.40	**99.88**
$Opt_{V_{15}E_{14}}$	99.40	99.40	**99.88**	99.40	99.40	**99.88**
horner_bs	99.30	99.30	**99.86**	99.30	99.30	**99.86**
$No_{V_{18}E_{18}}$	33.28	**99.50**	33.32	33.28	99.50	**99.90**
$Opt_{V_{25}E_{24}}$	99.30	99.30	99.30	99.30	99.30	99.30
arf	**99.10**	66.23	49.96	**99.10**	66.27	71.30
conv3	**66.31**	49.75	59.92	66.31	**98.71**	77.59
$No_{V_{30}E_{33}}$	33.25	**98.91**	49.94	33.25	**98.91**	66.53
$No_{V_{43}E_{50}}$	**99.10**	49.80	33.30	**99.10**	49.80	49.93
$Opt_{V_{59}E_{68}}$	45.24	39.86	**59.89**	45.24	49.70	**59.89**
matmul	**57.36**	28.84	29.95	**57.36**	53.82	49.87
invert_matrix	12.56	**19.86**	17.85	12.56	**39.12**	35.48

Considerando que o algoritmo YOTO apresenta, em média, um tempo de execução cerca de 100 vezes mais rápido do que SA, a comparação foi estabelecida entre seis instâncias do SA e 600 instâncias do YOTO. Para o algoritmo YOTT, cuja profundidade de *pipeline* é maior (dez estágios), foram executadas 100 instâncias. Os valores destacados em negrito indicam o melhor desempenho de vazão de dados observado para cada DFG. Rendimentos superiores a 99,0% são considerados praticamente ideais, uma vez que a diferença residual em relação a 100% decorre da latência de inicialização do fluxo de dados na arquitetura CGRA.

A análise dos resultados indica que, em cinco dos doze DFGs avaliados, os três algoritmos produziram posicionamentos equivalentes em termos de vazão. Observa-se, ainda, que o algoritmo YOTO apresentou o melhor desempenho em um dos circuitos, enquanto os algoritmos SA e YOTT demonstraram superioridade em outros três casos distintos.

Esses resultados indicam que, embora algoritmos baseados em travessia, como o YOTO, produzam soluções menos otimizadas sob a perspectiva do comprimento total das interconexões, tais abordagens são capazes de proporcionar vazões equivalentes ou até superiores em diversos cenários. Além disso, ao considerar que a unidade YOTO ocupa aproximadamente dez vezes menos área em comparação com a unidade baseada no algoritmo SA, sua adoção mostra-se particularmente vantajosa em aplicações que exigem elevada eficiência em termos de área de implementação, sem comprometer de forma significativa o desempenho global do sistema.

4.2 SA, YOTO e YOTT como posicionadores para FPGAs

Nesta seção, é apresentada a avaliação experimental dos algoritmos SA, YOTO e YOTT para arquiteturas FPGA baseadas em LUTs. A análise adota como referência a ferramenta VPR [11], inicialmente em sua versão 5 e, posteriormente, na versão 9.

Inicialmente, na Seção 4.2.1, são descritas as arquiteturas-alvo empregadas, os fluxos de conversão dos *benchmarks* e os conjuntos experimentais considerados (MCNC20, EPFL e, em etapa posterior, MNIST e JSC). Em seguida, a Seção 4.2.2 apresenta as estratégias de travessia dos grafos, bem como as configurações adotadas no VPR.

A Seção 4.2.3 discute os resultados iniciais em termos de caminho crítico e tempo de execução comparando com o VPR. Na sequência, a Seção 4.2.4 investiga o impacto do uso de memória *cache* sobre o desempenho dos aceleradores.

A Seção 4.2.5 analisa as características estruturais dos grafos e sua influência nos algoritmos, particularmente em cenários de alta ocupação da matriz. Por fim, a Seção 4.2.6 apresenta os resultados de uma estratégia híbrida de busca com o objetivo de reduzir o número de tentativas de alocação e melhorar a eficiência global do posicionamento.

4.2.1 Arquitetura-alvo e *benchmarks*

Para a utilização da ferramenta VPR, tornou-se necessária a especificação das arquiteturas-alvo no formato XML, de modo a contemplar a descrição detalhada dos blocos lógicos, das interconexões e dos pinos de entrada e saída. Nesse contexto, foram desenvolvidas quatro arquiteturas distintas, com diferentes níveis de granularidade, baseadas em LUTs de 3, 4, 5 e 6 entradas, as quais foram adotadas como referência nos experimentos realizados.

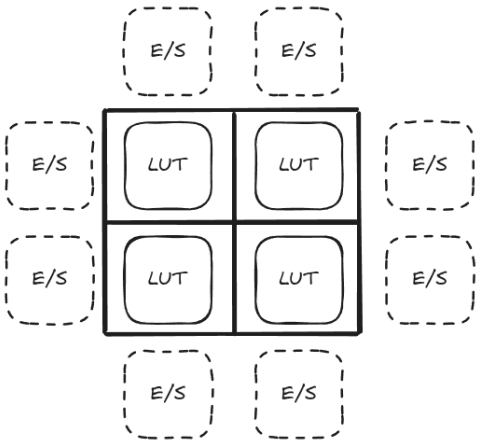


Figura 4.2: Exemplo de arquitetura básica para FPGA com 2x2 LUTs e 8 pinos de entrada/saída.

A Figura 4.2 ilustra a estrutura de uma arquitetura 2×2 composta por LUTs, pinos de entrada e saída distribuídos ao longo das bordas e barramentos internos destinados ao roteamento de sinais. Inicialmente, os aceleradores foram projetados para o posicionamento de um nó de E/S por célula de E/S. Posteriormente, a quantidade de nós de E/S foi ampliada para um melhor aproveitamento estrutura da matriz de posicionamento e compactação do circuito posicionado.

Observa-se que as regiões de quina não dispõem de pinos de E/S. O código em XML correspondente à geração dessa arquitetura para o VPR 5 encontra-se disponível no Apêndice A. Posteriormente, passou-se a utilizar o VPR 9. Para essa versão, tornou-se necessária a reescrita da linguagem de descrição da arquitetura do FPGA, em função das alterações introduzidas na linguagem utilizada. O mesmo exemplo encontra-se apresentado no Apêndice B.

Para a utilização dos *benchmarks*, fez-se necessária a realização de algumas etapas de conversão, uma vez que os arquivos originais estão descritos na linguagem Verilog. Para tanto, empregou-se a ferramenta ABC [60] para a conversão dos circuitos para o formato BLIF. A partir desse formato intermediário, os circuitos foram convertidos para dois outros formatos: DOT, utilizados tanto para visualização quanto como entrada dos algoritmos de posicionamento desenvolvidos; e NET, empregado como entrada para o VPR. Após o posicionamento dos circuitos pelos algoritmos propostos, arquivos no formato PLACE são gerados de acordo com as especificações do VPR [81]. Exemplos dos arquivos nos formatos NET e PLACE encontram-se disponíveis no Apêndice C.

Os experimentos iniciais conduzidos com os conjuntos de *benchmarks* MCNC20 e EPFL possibilitaram uma análise comparativa dos algoritmos propostos em relação à ferramenta VPR v5 e v9. Os *benchmarks* do conjunto MCNC20, por corresponderem a circuitos de menor complexidade, foram avaliados com diferentes arquiteturas baseadas em LUTs de 3, 4, 5 e 6 entradas, o que permite a comparação dos resultados obtidos em distintos níveis de granularidade lógica.

Por sua vez, os *benchmarks* do conjunto EPFL foram avaliados exclusivamente em arquiteturas baseadas em LUTs de seis entradas e comparados com os resultados reportados pelo VPR v5 e v9 para essa mesma configuração. Em uma etapa posterior, foram utilizados também outros dois *benchmarks* , MINIST e JSC [44], por serem empregados em abordagens de aprendizado de máquina baseadas em LUTs [28, 72, 6, 55, 9, 7, 39, 71, 8, 44, 94, 56, 77, 18].

As principais características dos circuitos pertencentes aos conjuntos MCNC20 e EPFL com LUTs de seis entradas encontram-se organizadas na Tabela 4.10, dispostas em ordem crescente segundo o número de LUTs. Para os *benchmarks* MINIST e JSC, as características se encontram na Tabela 4.11.

Uma das principais diferenças entre os dois conjuntos de *benchmarks*, MCNC20

Tabela 4.10: Dados dos *Benchmarks* MCNC20 e EPFL (LUT 6)

MCNC20				**EPFL**			
Benchmark	**LUT**	**Arestas**	**E/S**	***Benchmark***	**LUT**	**Arestas**	**E/S**
too-lrg	103	479	41	ctrl	27	154	32
ex5p	578	2837	65	int2float	49	261	18
apex4	777	3588	27	router	75	348	63
alu4	829	3932	22	cavlc	118	634	21
misex3	846	3913	28	priority	194	922	136
apex2	1019	4637	41	adder	243	1140	385
ex1010	2442	11319	20	dec	287	940	264
				i2c	300	1500	263
				bar	512	2816	263
				max	763	3325	642
				sin	1475	6781	49
				voter	1548	7648	1002
				square	3933	16536	190
				sqrt	4784	21844	192
				multiplier	4815	22688	228
				log2	7906	36192	64
				div	9883	40840	256

e EPFL, está na quantidade de arestas. Enquanto os circuitos do conjunto MCNC20 apresentam uma razão relativamente equilibrada entre o número de nós e de conexões, alguns circuitos do conjunto EPFL, como o *div*, atingem valores substancialmente mais elevados, com aproximadamente 10 mil nós e 30 mil arestas. Essa elevada densidade de interconexões tende a diminuir o desempenho do roteador, uma vez que circuitos com maior grau de conectividade estão mais suscetíveis a congestionamentos durante o processo de roteamento, o que pode comprometer a qualidade da solução final.

Além disso, o número total de nós exerce influência direta tanto sobre o tempo de execução dos algoritmos quanto sobre a ocupação da matriz de posicionamento, o que o caracteriza como um parâmetro fundamental para a avaliação do tempo de execução e da escalabilidade das arquiteturas analisadas.

Tabela 4.11: ML Datasets Mapped on Circuit using TreeLut [44].

Benchmark	**LUT**	**Arestas**	**E/S**	**Precisão**	**Parâmetros**
mnist1	4,102	16629	1,954	95	[30, 4, 0.8, 4, 4]
mnist2	4,421	18216	1,888	95	[30, 4, 0.8, 4, 4, Arg]
mnist3	5,990	5990	2,052	96	[30, 5, 0.8, 4, 4]
mnist4	6,385	6385	1,986	96	[30, 5, 0.8, 4, 4, Arg]
jsc1	940	4140	145	74	[10, 5, 0.3, 8, 2]
jsc2	2,686	11318	123	75	[13, 5, 0.8, 8, 4, Arg]

As primeiras abordagens de redes neurais baseadas em LUTs demandavam até 100,000 LUTs. Trabalhos mais recentes reduziram esse requisito para a ordem de 10,000 LUTs [44, 9, 8]. Neste trabalho, tais dimensões de circuito são geradas por meio da abordagem baseada em árvores apresentada em [44].

A Tabela 4.11 apresenta quatro implementações para o conjunto de dados MNIST e duas para o conjunto JSC. A qualidade é medida pela a coluna **acurácia** e o custo pela coluna **LUTs**. A principal diferença entre MNIST 1–2 e MNIST 3–4 está na profundidade das árvores, que são, respectivamente, 4 e 5. Em cada grupo, a distinção entre as versões 1–2 e 3–4 refere-se à presença do circuito de *argmax* na saída. As versões 1 e 3 apresentam dez saídas de 4 bits por dígito, enquanto as versões 2 e 4 incorporam a votação por maioria (*argmax*), e o resultado é uma única palavra de 4 bits que representa o dígito mais provável.

O vetor de parâmetros apresentado na coluna **Parâmetros** possui a forma **[n_trees, max_depth, eta, w_feature, w_tree, módulo *argmax*]**. Os parâmetros *w_feature* e *w_tree* são específicos da ferramenta TreeLUT [44] e definem, respectivamente, a quantização das características de entrada e dos votos das árvores. Observa-se que a inclusão do módulo *argmax* exerce impacto reduzido no tamanho do circuito. Por outro lado, o aumento da profundidade das árvores de 4 para 5 eleva a acurácia em aproximadamente 1% e acarreta um aumento de aproximadamente 40% na quantidade de LUTs, o que aumenta a área necessária para o posicionamento.

Para conjunto de dados JSC, foram mapeadas duas versões distintas. A primeira versão com 10 árvores com profundidade 5, parâmetro *eta* igual a 0,5, características de entrada quantizadas em 8b e peso de voto das árvores representado com apenas 2b. A segunda versão aprimora a precisão por meio do uso de 13 árvores e do aumento da largura do peso de voto das árvores para 4 bits, além da inclusão do módulo *argmax*, responsável pelo cálculo da classe majoritária entre as cinco classes consideradas.

4.2.2 Estratégias de travessia do grafo para os algoritmos YOTO e YOTT

Os algoritmos YOTO e YOTT requerem, como entrada, uma lista de arestas para a realização do posicionamento. No caso específico do YOTO, a ordem na qual essas arestas são fornecidas influencia diretamente a solução obtida. Tal característica decorre da natureza gulosa do algoritmo, que privilegia, a cada iteração, a escolha de uma LUT adjacente que minimize a distância entre os nós conectados.

Dessa forma, torna-se fundamental que a lista de arestas preserve uma ordenação coerente com a estrutura do grafo, de modo que arestas associadas a relações topologicamente próximas sejam processadas de forma sequencial. Essa organização aumenta a probabilidade de formação de regiões mais compactas e coerentes na malha, além de

favorecer soluções caracterizadas por menor dispersão espacial.

O algoritmo YOTT adota uma estratégia distinta, fundamentada na identificação prévia de convergências no grafo. Nessa etapa inicial, são computadas, para cada aresta, informações métricas relacionadas às distâncias entre o nó a ser posicionado e um conjunto de nós considerados relevantes, incluindo nós de entrada, de saída e aqueles conectados por relações estruturais. A partir dessas métricas, o YOTT realiza a alocação dos elementos em LUTs de modo a minimizar o custo global do grafo e priorizar configurações espaciais que mantenham, de forma balanceada, a proximidade em relação aos múltiplos destinos. Essa formulação possibilita uma atuação de caráter mais global ao considerar a topologia do circuito como um todo.

Para a execução do YOTO, foram empregados dois métodos distintos de ordenação das arestas do circuito: um baseado em busca em profundidade (DFS) tradicional e outro, também fundamentado em DFS, porém com priorização explícita do caminho crítico. No caso do YOTT, a ordenação das arestas foi realizada por meio do algoritmo em *zigzag* [14], o qual identifica convergências no grafo e viabiliza a inserção sistemática de anotações.

Cada algoritmo foi executado mil vezes. A partir dos resultados obtidos, foram selecionadas, para cada instância, as dez soluções que apresentaram os menores custos totais de roteamento, calculados pela soma das distâncias de *Manhattan*. Em seguida, foi computada a média desses valores para fins de análise comparativa. Primeiramente, foi utilizado o VPR v5 a configuração padrão e *fast* dos parâmetros disponibilizada pela ferramenta. Posteriormente, os experimentos foram repetidos com o VPR v9, utilizando as configurações de posicionamento *bounding box* e *critical timing*. A configuração *bounding box* prioriza a minimização do comprimento total das interconexões, enquanto a configuração *critical timing* busca reduzir simultaneamente o comprimento dos fios e o caminho crítico do circuito.

4.2.3 Qualidade da Solução e Tempo de Execução

Inicialmente, avaliamos a qualidade das soluções em comparação com a ferramenta VPR. O critério foi o comprimento do caminho crítico e também foram medidos os tempos de execução dos algoritmos. A metodologia empregada nos experimentos pode ser vista na Figura 4.3.

Em uma primeira etapa, a ferramenta VPR (v5 e v9) foi executada com seus parâmetros padrão, restrita à fase de posicionamento, de modo a viabilizar uma comparação direta com os tempos de execução dos algoritmos desenvolvidos. Em seguida, o VPR foi novamente empregado para a realização do roteamento dos posicionamentos por ele gerados, o que possibilitou a obtenção do valor do caminho crítico, adotado como referência para fins de comparação.

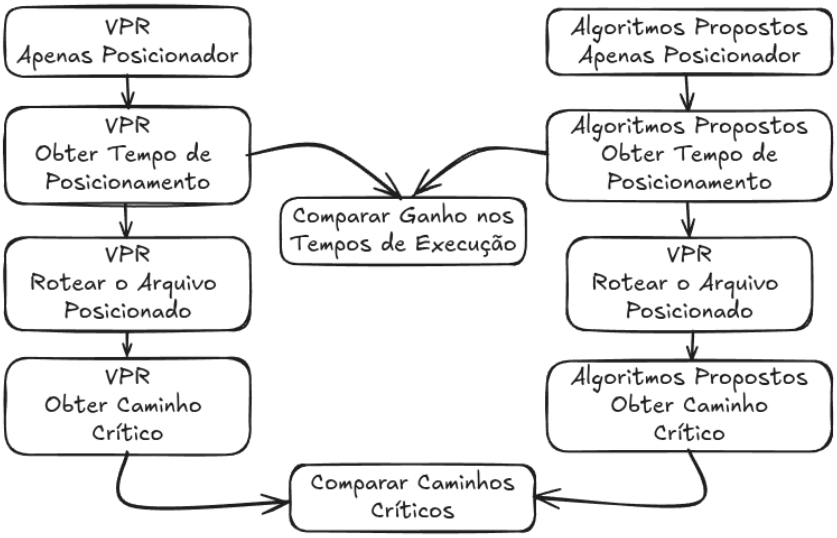


Figura 4.3: Passos para a execução dos experimentos

Para a medição do tempo de execução, deve-se considerar que a operação ocorre em regime de *pipeline*, conforme ilustrado na Figura 4.4 (a), que apresenta o método adotado para a aferição do tempo de execução em *hardware*. Considere-se um *pipeline* composto por quatro estágios, no qual são executadas quatro *threads*. As duas primeiras *threads* percorrem o *pipeline* três vezes, enquanto as duas últimas realizam quatro passagens completas.

Uma vez que a execução das *threads* ocorre de forma sobreposta ao longo dos estágios do *pipeline*, o tempo total não corresponde à soma direta dos tempos individuais. Nessas condições, pode-se afirmar que, em média, cada *thread* demanda 3,5 ciclos de *clock* para completar sua execução, desde que o tempo associado ao preenchimento inicial do *pipeline* seja desconsiderado.

Entretanto, deve-se considerar que, para a obtenção do tempo de execução, é necessária a execução simultânea de quatro *threads*. Dessa forma, a execução de uma ou de quatro *threads* resulta em valores de latência muito próximos. Então, adotou-se como tempo de execução a latência correspondente ao processamento de quatro *threads*, que demora 19 ciclos. Considerando-se um período de *clock* de 4 ns, correspondente a uma frequência de operação de 250 MHz, esse valor pode ser considerado conservador para dispositivos FPGA nas implementações dos algoritmos YOTO e YOTT. No caso do algoritmo SA, esse valor é duplicado, uma vez que sua execução ocorre a cada dois ciclos, em função da restrição de permitir apenas uma operação de escrita por ciclo.

A Figura 4.4 (b) ilustra o cenário no qual se emprega a memória *cache*. Na ocorrência de uma falha de acesso, a *thread* deixa de avançar em uma ou mais iterações, de modo que o impacto dessas falhas passa a compor o tempo total de execução. No exemplo apresentado, observam-se quatro falhas ao longo de dezoito iterações $(4 + 4 + 5 + 5)$ distribuídas entre as quatro *threads*, o que corresponde a uma taxa de falhas de 22%.

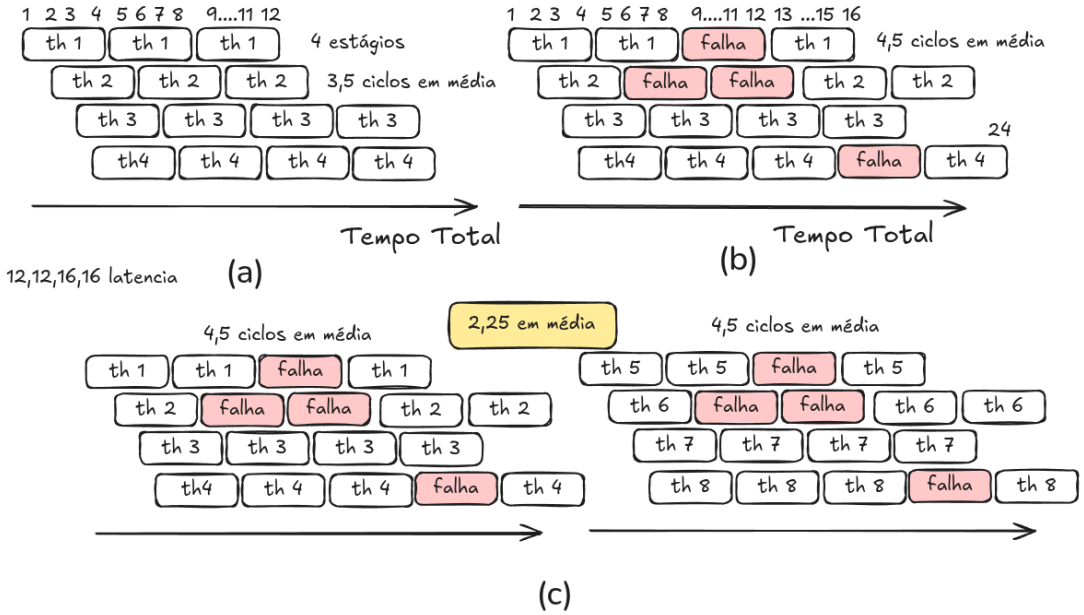


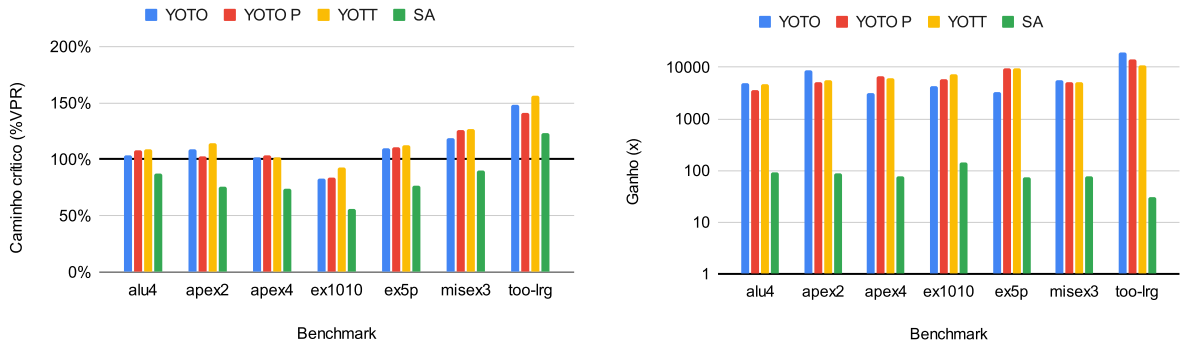
Figura 4.4: (a) Execução de 4 *threads* simbólica sem *cache*; (b) Execução com 4 *threads*, *cache* e falhas; (c) Execução com duas unidades com 8 *threads*, *cache* e falhas.

Como consequência, o número médio de ciclos por *thread* eleva-se para 4,5 ciclos, e o tempo total de execução aumenta de 19 para 24 ciclos, o que representa um acréscimo de aproximadamente 1,26×.

A principal vantagem do uso de memória *cache* consiste na possibilidade de aumentar o número de unidades de execução disponíveis. Com duas unidades, conforme ilustrado na Figura 4.4 (c), obtém-se uma redução do tempo médio por *thread* para 2,25 ciclos. Mais relevante, contudo, é o fato de que, nessa configuração, passam a existir oito tentativas simultâneas para a seleção da melhor alternativa de posicionamento, sem aumento do tempo total de execução, que permanece em 24 ciclos. Essa característica possibilita a obtenção de soluções com menor caminho crítico e redução do tempo de execução.

Inicialmente, para a avaliação dos posicionamentos gerados, após a produção dos arquivos de posicionamento, estes foram submetidos ao roteador da ferramenta VPR v5, de modo a assegurar a uniformidade do procedimento experimental. Dessa forma, os caminhos críticos resultantes do roteamento dos posicionamentos produzidos pelos algoritmos SA, YOTO e YOTT foram obtidos de maneira compatível com aqueles gerados diretamente pelo VPR v5, o que garante a equivalência e a consistência na comparação dos mesmos. Posteriormente, os experimentos também foram conduzidos com a versão 9 do VPR. Primeiramente, são apresentados e analisados os resultados em comparação com a versão 5 e, em seguida, discute-se os resultados obtidos com a versão 9.

A Figura 4.5 apresenta uma análise comparativa entre os algoritmos propostos e o VPR v5, considerando dois critérios principais: (i) caminho crítico obtido da



(a) Caminho crítico em relação ao VPR v5 para LUT 3 para a melhor solução de posicionamento.

(b) Tempo de execução dos posicionadores em relação ao VPR v5 para LUT 3.

Figura 4.5: Caminho crítico das soluções versus ganho no tempo de execução (LUT 3). *Benchmark* MCNC20

solução e (ii) ganho em tempo de execução em relação ao VPR v5. Na Figura (a), são apresentados os caminhos críticos referentes aos posicionamentos produzidos pelo algoritmo, quantificados pela razão entre o caminho crítico estimado e aquele obtido pelo VPR v5. O valor de 100% corresponde à solução de referência, isto é, ao resultado gerado pelo VPR v5. Valores inferiores a 100% indicam soluções de maior caminho crítico, enquanto valores superiores a esse limiar indicam uma piora no tempo de execução em comparação à referência adotada. A Figura (b), por sua vez, apresenta o ganho em tempo de execução, no qual o valor unitário representa um tempo de execução equivalente ao do VPR v5, e valores superiores indicam aceleração em relação a essa ferramenta para o melhor posicionamento de mil execuções.

O tempo de execução do VPR v5 situa-se na ordem de 10 a 20 segundos para os exemplos considerados, restrito à etapa de posicionamento. A adoção de aceleradores em *hardware* reduz esse tempo para a ordem de milissegundos, o que viabiliza, inclusive, a aplicação de estratégias de reconfiguração dinâmica em plataformas reconfiguráveis, capazes de substituir o circuito durante a própria execução.

Entre os algoritmos avaliados, o SA apresentou o melhor caminho crítico das soluções, superando o VPR v5 na maioria dos *benchmarks* analisados, com exceção do circuito *too-lrg*. Em contrapartida, esse algoritmo apresentou o menor ganho em tempo de execução quando comparado aos demais métodos propostos, embora o ganho observado varie entre uma e duas ordens de grandeza em relação ao VPR v5.

Cabe ressaltar que, neste conjunto de experimentos, o SA foi avaliado em arquiteturas baseadas em LUTs com três entradas. O aumento do número de entradas das LUTs para quatro, cinco ou seis impacta diretamente o custo de *hardware* do algoritmo, uma vez que essa modificação influencia os requisitos de memória associados ao armazenamento das listas de adjacência.

Os algoritmos YOTO e YOTT apresentaram resultados semelhantes no que se refere ao comprimento dos caminhos críticos das soluções, com uma degradação moderada em comparação ao VPR v5 na maioria dos *benchmarks* avaliados. As soluções obtidas por esses algoritmos exibiram caminhos críticos que variaram entre valores ligeiramente superiores a 100% e até aproximadamente 150% daqueles produzidos pelo VPR v5. Em contrapartida, ambos os algoritmos proporcionaram ganhos expressivos em tempo de execução, que variaram entre três e quatro ordens de magnitude em relação ao VPR v5. Esses resultados evidenciam o potencial dessas abordagens para cenários de aplicação nos quais a eficiência computacional constitui o principal critério de otimização.

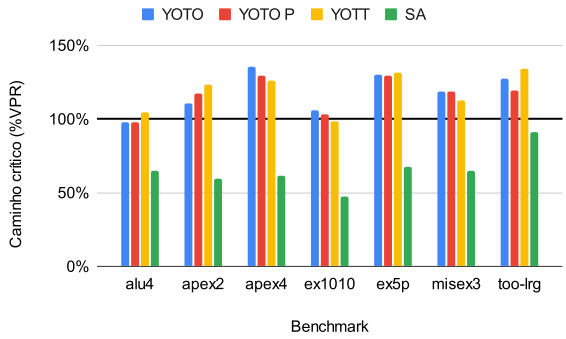
O impacto da ordenação das arestas nos resultados obtidos para o algoritmo YOTO foi avaliado por meio de duas variantes do algoritmo de busca em profundidade (DFS): uma implementação convencional e outra com priorização explícita do caminho crítico. Os resultados obtidos para ambas as abordagens foram semelhantes, o que indica que a ordenação das arestas não exerce influência significativa sobre o caminho crítico das soluções produzidas pelo algoritmo.

Por outro lado, o algoritmo YOTT apresentou um resultado inferior em termos de comprimento do caminho crítico das soluções, mesmo com a utilização de anotações estruturais destinadas a orientar o processo de posicionamento. Esse resultado indica que a estratégia de anotações adotada não produziu os efeitos esperados, o que abre espaço para a formulação de novas hipóteses no contexto da abordagem proposta. Ao priorizar reconvergências locais com base nas anotações, o YOTT tende a reduzir o número total de recursos de roteamento empregados; entretanto, essa priorização ocorre em detrimento da consideração explícita do caminho crítico, o que compromete a solução entregue.

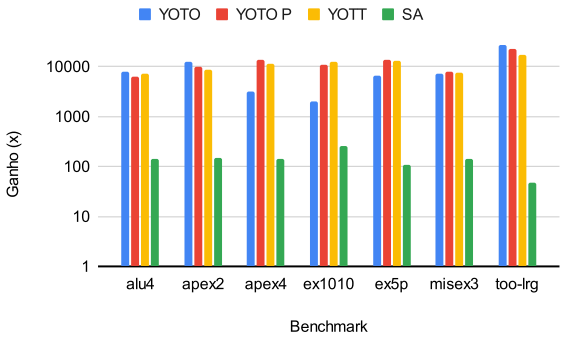
A Figura 4.6 apresenta os resultados complementares dos experimentos realizados com arquiteturas baseadas em LUTs de 4, 5 e 6 entradas. As subfiguras (a) e (b) mostram os resultados obtidos com LUT de 4 entradas; as subfiguras (c) e (d) apresentam os resultados correspondentes às arquiteturas com LUT de 5 entradas; e, por fim, as subfiguras (e) e (f) exibem os resultados referentes às arquiteturas baseadas em LUT de 6 entradas.

De modo geral, observa-se a preservação do padrão previamente identificado na Figura 4.5. O algoritmo SA continua a apresentar soluções comparáveis ou superiores àquelas obtidas pelo posicionador de referência VPR v5, ainda que com ganhos mais modestos em tempo de execução, os quais variam entre uma e duas ordens de grandeza. Em contraste, os algoritmos YOTO e YOTT mantêm ganhos expressivos em tempo de execução, situados entre três e quatro ordens de grandeza em relação ao VPR v5, ao custo de uma degradação do caminho crítico que pode atingir até 50% em determinados cenários.

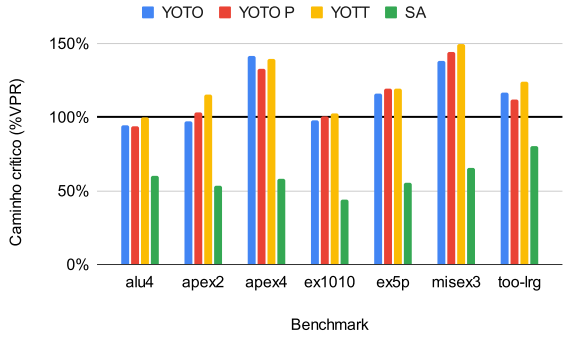
A Figura 4.7 apresenta os resultados obtidos para o conjunto de *benchmarks* EPFL



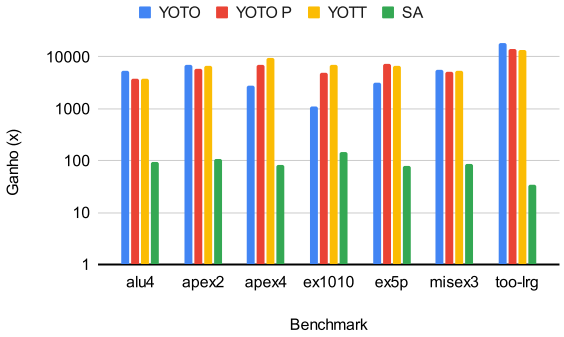
(a) Caminho crítico em relação ao VPR v5 para LUT 4, melhor de 1000 soluções.



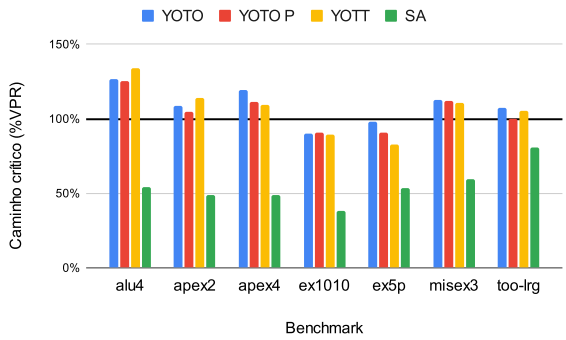
(b) Tempo de execução dos posicionadores em relação ao VPR v5 para LUT 4.



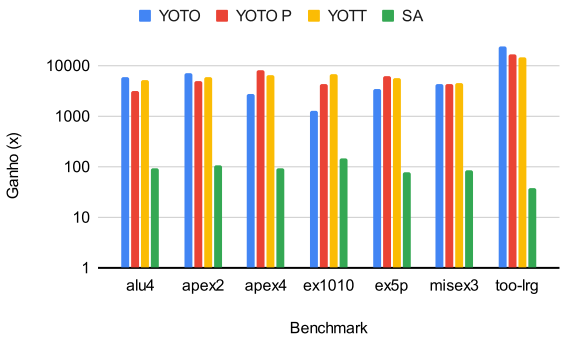
(c) Caminho crítico em relação ao VPR v5 para LUT 5 melhor de 1000 soluções.



(d) Tempo de execução dos posicionadores em relação ao VPR v5 para LUT 5.



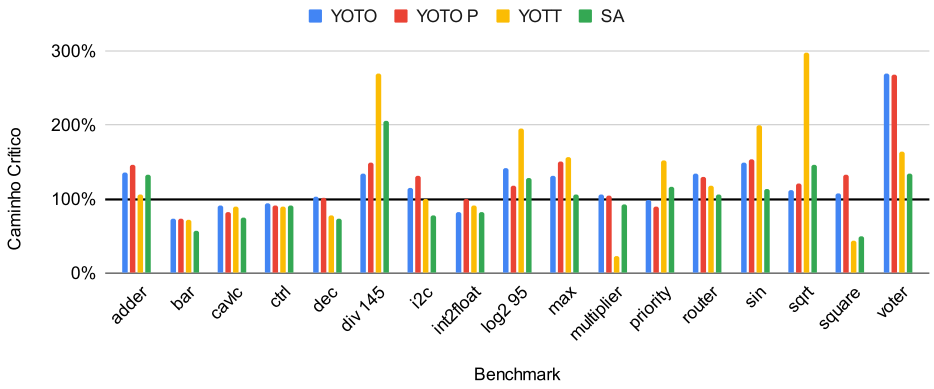
(e) Caminho crítico em relação ao VPR v5 para LUT 6, melhor de 1000 soluções.



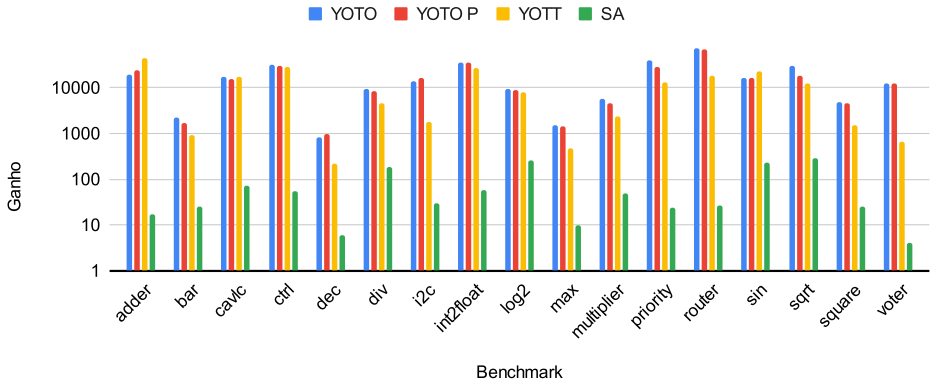
(f) Tempo de execução dos posicionadores em relação ao VPR v5 para LUT 6.

Figura 4.6: Caminho crítico das soluções versus o tempo de execução (LUTs 4, 5, e 6). *Benchmark* MCNC20.

em uma arquitetura baseada em LUTs de 6 entradas. A subfigura (a) mostra o caminho crítico das soluções de posicionamento, quantificada pela razão entre o caminho crítico produzido por cada algoritmo e aquele obtido pelo VPR v5, adotado como referência (100%). A subfigura (b) apresenta os ganhos em tempo de execução em relação ao VPR v5.



(a) Caminho crítico em relação ao VPR v5 para melhor solução em 1000 posicionamentos.



(b) Tempo de execução dos posicionadores em relação ao VPR v5.

Figura 4.7: Caminho crítico das soluções versus o tempo de execução (LUT 6). *Benchmark* EPFL

No caso dos circuitos do conjunto EPFL, caracterizados por maior complexidade estrutural e um maior número de componentes, observa-se que a degradação no caminho crítico dos posicionamentos produzidos pelos algoritmos propostos permanece semelhante àquela verificada nos experimentos com o conjunto MCNC20. Entretanto, identifica-se uma limitação mais evidente em relação ao algoritmo YOTT. Os resultados indicam que a estratégia atualmente adotada para o sequenciamento das arestas e para o uso de anotações estruturais, empregada como mecanismo de orientação do posicionamento, não tem se mostrado eficaz na melhoria das soluções. Esse comportamento aponta para a necessidade de reformulação ou aperfeiçoamento dessas estratégias, com ênfase na adaptação do algoritmo YOTT a circuitos de maior escala e maior complexidade topológica, de modo a torná-lo mais competitivo.

Uma exceção a esse padrão geral foi observada no circuito *multiplier*, no qual o YOTT superou de forma significativa os demais algoritmos ao produzir uma solução melhor do que as obtidas tanto pelo YOTO quanto pelo SA. Todavia, em outros casos, como os circuitos *div* e *sqrt*, o algoritmo apresentou degradações superiores a 200%.

Desconsiderada essa exceção pontual, os resultados do YOTT mantêm-se, em geral, comparáveis ou inferiores aos alcançados pelo YOTO, o que reforça a hipótese de que os mecanismos internos do YOTT ainda demandam aprimoramentos para lidar de forma mais eficaz com circuitos de grande porte e elevada complexidade estrutural.

Cabe destacar que todas as execuções do VPR v5 foram realizadas com a utilização de seus parâmetros padrão, os quais são configurados para privilegiar a obtenção de soluções com redução no caminho crítico, ainda que à custa de elevado esforço computacional. Essa estratégia, embora eficaz, apresenta limitações práticas de escalabilidade.

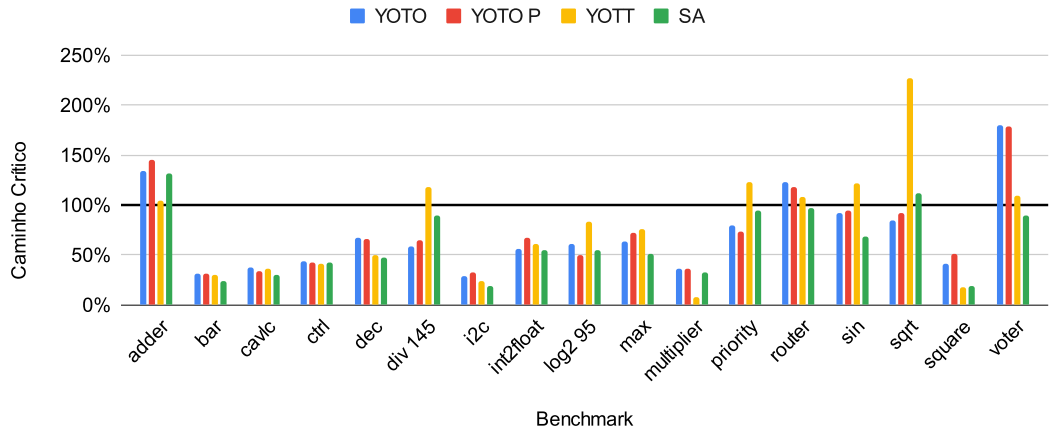
Como exemplo, para o *benchmark square*, o tempo total de execução, considerando conjuntamente as etapas de posicionamento e roteamento, ultrapassou onze dias. Trata-se, contudo, de um caso atípico, uma vez que o VPR v5 entra em um processo iterativo de otimização, no qual múltiplas soluções de posicionamento e roteamento são avaliadas com o objetivo de selecionar a de menor utilização de recursos do FPGA e o menor caminho crítico. Para a maior parte dos demais circuitos analisados, o tempo total de execução situa-se na faixa de segundos a poucos minutos.

Alterações nos parâmetros do VPR v5, com o objetivo de acelerar a execução por meio da simplificação de heurísticas internas ou da redução do número de iterações, resultaram em perdas significativas no comprimento do caminho crítico das soluções produzidas. Ainda assim, tais ajustes permitiram reduzir o tempo total de posicionamento e roteamento para aproximadamente nove horas.

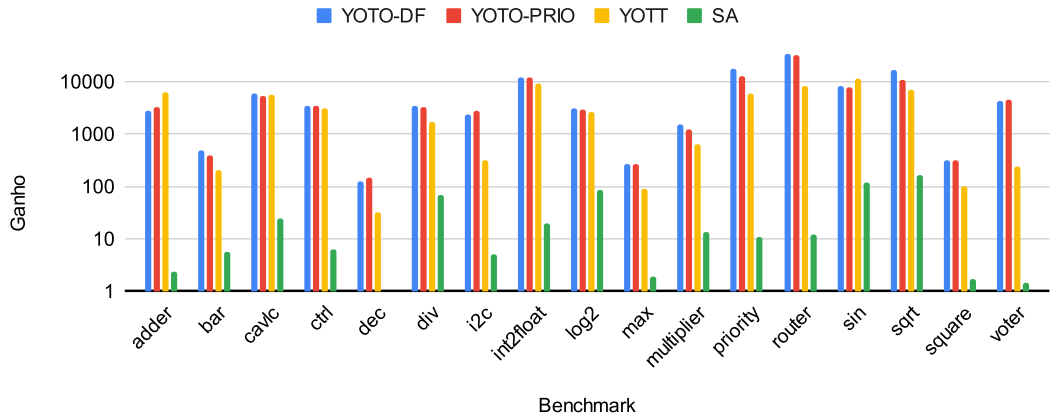
A Figura 4.8 (a) ilustra a degradação do caminho crítico das soluções geradas pelo VPR v5 quando configurado com parâmetros voltados à redução do tempo de execução, em comparação com os resultados obtidos pelos algoritmos desenvolvidos nesta pesquisa. Essa análise mostra, de forma mais explícita, o custo associado à obtenção de melhores soluções por meio do VPR v5.

Além da degradação observada nos caminhos críticos, identifica-se também uma redução no tempo de execução do VPR v5 quando comparado aos algoritmos propostos. Esse efeito pode ser observado pela diminuição dos ganhos de tempo apresentados nos gráficos em (b), os quais passam a variar de poucas vezes até, em casos específicos, aproximadamente dez mil vezes, valor que, nas execuções com os parâmetros padrão, ocorria de forma mais recorrente. Destaca-se, por exemplo, que, para o circuito *dec*, o algoritmo SA apresentou um tempo de execução superior ao do VPR v5 configurado com parâmetros otimizados.

Esse comportamento manifesta-se, de modo geral, com maior frequência em circuitos que apresentam baixa ocupação da área lógica. Nesses casos, a redução da complexidade espacial favorece a identificação de soluções pelo VPR v5 otimizado, que resulta em um menor tempo de execução. Os resultados indicam que, embora a otimização voltada à redução do tempo de execução torne o VPR v5 mais competitivo do ponto de vista do desempenho computacional, tal melhoria ocorre em detrimento



(a) Caminho crítico em relação ao VPR em execução rápida, considerando o melhor resultado de 1000 posicionamentos.



(b) Tempo de execução dos posicionadores em relação ao VPR em execução rápida.

Figura 4.8: Caminho crítico das soluções versus o tempo de execução (LUTs 6 em execução rápida). *Benchmark* EPFL

do caminho crítico das soluções produzidas, o que pode ser crucial em aplicações sensíveis à frequência máxima de operação.

Ressalta-se que o fluxo de execução dos experimentos seguiu o mesmo procedimento metodológico apresentado na Figura 4.3, com diferença apenas nos parâmetros ajustados do VPR v5, os quais foram configurados nesta etapa com o objetivo de acelerar os processos de posicionamento e roteamento.

Observa-se, adicionalmente, que, com exceção dos *benchmarks adder*, *router* e *voter*, todos os demais circuitos apresentaram melhores caminhos críticos quando processados pelos algoritmos propostos. Entretanto, ao se considerar como referência os resultados obtidos com a configuração padrão do VPR v5, verifica-se que todos os algoritmos analisados produziram soluções superiores àquelas geradas pela versão otimizada do VPR v5.

Além das análises isoladas de tempo de execução e de qualidade das soluções, foi conduzida uma avaliação com o objetivo de mensurar a relação custo-benefício associada à aplicação dos algoritmos propostos em comparação ao VPR v5. Essa avaliação tem por finalidade definir uma métrica composta que integre, de forma ponderada, o tempo de execução e o caminho crítico obtido, de modo a permitir uma comparação mais abrangente e informativa entre os métodos analisados.

Para esse fim, adotou-se a seguinte equação para o cálculo do custo relativo de execução:

$$\text{Custo Relativo} = \frac{T_{alg}}{T_{VPR}} \times \frac{C_{alg}}{C_{VPR}}$$

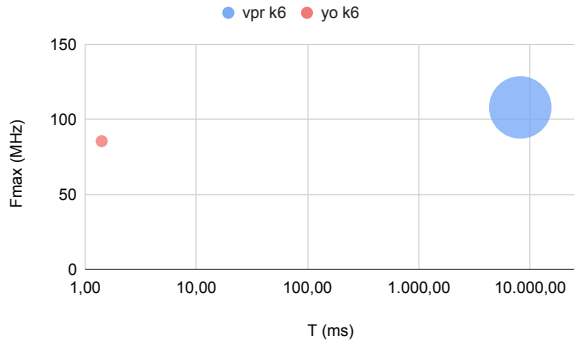
onde:

- T_{alg} = tempo de execução do algoritmo analisado;
- T_{VPR} = tempo de execução do VPR v5;
- C_{alg} = caminho crítico obtido pelo algoritmo analisado;
- C_{VPR} = caminho crítico obtido pelo VPR v5;

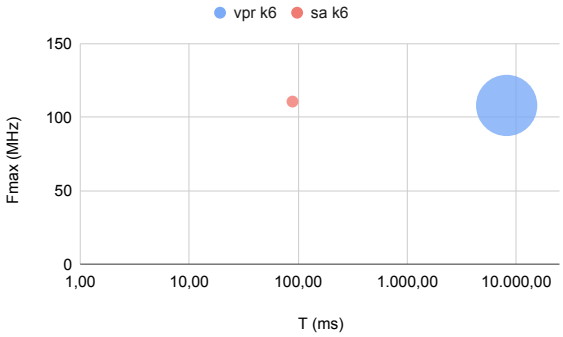
Essa métrica permite avaliar o impacto combinado do tempo de execução computacional e do caminho crítico das soluções obtidas. Valores inferiores a 1 indicam que o algoritmo proposto apresenta eficiência global superior à do VPR v5, ao considerar simultaneamente o tempo de execução e o caminho crítico. Em contraste, valores superiores a 1 caracterizam um custo global mais elevado, decorrente da perda de qualidade, do aumento do tempo de execução ou da combinação de ambos os fatores. Valores próximos a 1 indicam equivalência com o VPR v5 e refletem um equilíbrio entre os dois critérios considerados.

A Figura 4.9 apresenta a relação entre o custo de execução dos algoritmos de posicionamento, a frequência máxima alcançada pelo circuito e o tempo de execução associado a cada abordagem, considerando arquiteturas baseadas em LUTs de 6 entradas. O diâmetro das circunferências representa o custo relativo, calculado de acordo com a métrica definida anteriormente: quanto maior o diâmetro, maior o custo associado à solução. Em (a), apresenta-se a comparação entre os algoritmos VPR v5 e YOTO; em (b), entre VPR v5 e SA; e em (c), são analisadas as execuções dos algoritmos YOTO, YOTT e SA, com foco no circuito *alu4* do conjunto MCNC20, sem a inclusão do VPR v5.

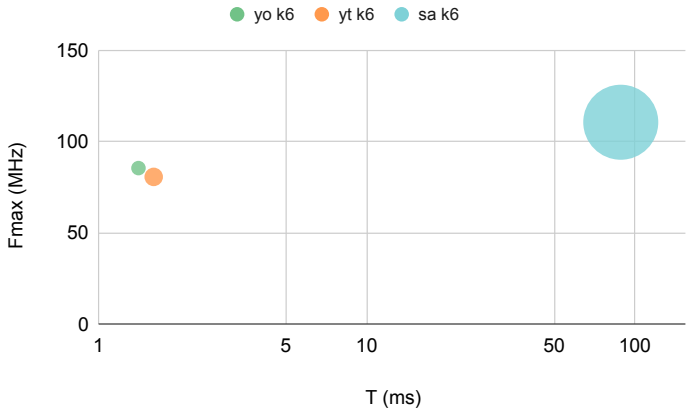
Observa-se, em (a) e (b), que, para arquiteturas baseadas em LUTs de menor cardinalidade, os algoritmos propostos tendem a produzir soluções com frequências máximas comparáveis às obtidas pelo VPR v5. Entretanto, à medida que o número de entradas das LUTs aumenta, essa similaridade é gradualmente reduzida. Tal comportamento



(a) Resultados versus custos dos algoritmos YOTO e VPR v5.



(b) Resultados versus custos dos algoritmos SA e VPR v5.



(c) Resultados versus custos dos algoritmos YOTO YOTT e SA.

Figura 4.9: Comparação entre os custos de execução dos algoritmos em relação ao VPR v5.

associa-se ao fato de que LUTs de maior capacidade reduz a complexidade estrutural do circuito, implicando um menor número de nós e arestas a serem posicionados, o que favorece a estratégia de busca empregada pelo VPR v5.

Apesar dessa vantagem relativa do VPR v5 em arquiteturas mais compactas, o tempo de execução é um fator determinante na comparação entre as abordagens. O algoritmo YOTO, por exemplo, apresenta acelerações da ordem de 1.000 a 10.000 vezes em relação ao VPR v5, enquanto o SA proporciona reduções no tempo de execução de aproximadamente duas ordens de magnitude.

Na análise apresentada em (c), observa-se a comparação direta entre os algoritmos YOTO, YOTT e SA, sem a referência ao VPR v5, o que possibilita uma avaliação isolada de suas características relativas. Verifica-se que o SA produz melhores soluções, evidenciado pelas frequências mais elevadas associadas às soluções obtidas, ainda que à custa de maior esforço computacional e tempos de execução superiores. Por outro lado, os algoritmos YOTO e YOTT apresentam valores semelhantes, embora o custo

relativo associado ao YOTT permaneça ligeiramente superior ao observado para o YOTO.

4.2.4 Execução com memória *cache*

Foram conduzidos experimentos com o emprego de memória *cache* como mecanismo de armazenamento intermediário para os dados associados às memórias N_2C e C_2N. A estratégia adotada baseou-se na utilização de estruturas de tamanho fixo, projetadas para manter temporariamente os dados mais frequentemente acessados ao longo da execução do algoritmo.

Dois experimentos distintos foram realizados com o objetivo de avaliar a influência do tamanho da *cache* sobre a taxa de acertos e, consequentemente, sobre a eficiência do uso de memória no contexto do posicionamento. As *caches* consideradas apresentaram capacidades de 256 e 4096 palavras, organizadas segundo a política de mapeamento direto e com 4 palavras por linha, o que resulta em 64 e 1024 linhas, respectivamente. Para cada configuração, foi quantificada a taxa de falhas de acesso (*cache misses*), de modo a possibilitar a análise da eficiência de reutilização de dados em função do tamanho da *cache*. Todos os experimentos foram realizados com o algoritmo YOTO, o qual adota uma estratégia de alocação de recursos que organiza os acessos à memória de forma a favorecer as localidades temporal e espacial dos dados.

Nos experimentos com os *benchmarks* MCNC20, foram consideradas arquiteturas baseadas em LUTs de 3, 4, 5 e 6 entradas. Para os *benchmarks* EPFL, as execuções restringiram-se à configuração com LUTs de 6 entradas, conforme discutido anteriormente. Os resultados obtidos para os *benchmarks* MCNC20 são apresentados nas Figuras 4.10 e 4.11, correspondentes, respectivamente, aos tamanhos de *cache* de 256 e 4096 palavras. Para os *benchmarks* EPFL, os resultados encontram-se nas Figuras 4.12 e 4.13. Ressalta-se que os gráficos foram gerados considerando todas as falhas de acesso às *caches*, inclusive aquelas associadas ao processo de inicialização.

A quantidade de bits necessários para a representação de uma célula depende da dimensão máxima da matriz associada a cada *benchmark*. Define-se como conjunto de trabalho o total de posições distintas da matriz de posicionamento que podem ser acessadas pelas estruturas N_2C e C_2N durante a execução do algoritmo. Assim, para cada circuito, o tamanho da palavra corresponde ao número mínimo de bits necessários para codificar todas as posições possíveis dessa matriz.

Como a *cache* é global, o conjunto de trabalho considera o total de posições potencialmente acessadas por todas as *threads*. A quantidade de células para cada *benchmark*, bem como o número de bits necessários para sua representação, encontram-se na Tabela 4.12, colunas **Células 10 *threads*** e **Bits**.

Assim, o tamanho de cada linha da *cache*, desconsiderando-se o campo *TAG*, é

dado por $N_{palavras} \times B + 1$ em que B representa o número de bits necessários para o endereçamento de uma célula da matriz e o termo adicional de 1 bit corresponde ao campo de validade da linha.

Como a quantidade de linhas das *caches* permaneceu fixa nos experimentos em 256 ou 4096 linhas, calculou-se a relação entre o tamanho necessário para o armazenamento completo do grafo e o tamanho da *cache*. As proporções podem ser vistas nas colunas % para as *caches* de 256 e 4096 palavras na Tabela 4.12. Observa-se que, para alguns *benchmarks*, como o *ctrl*, a capacidade nominal da *cache* de 256 palavras já se aproxima do tamanho estimado do conjunto de trabalho, o que explica a reduzida taxa de falhas observada nesses casos.

Tabela 4.12: Dimensão das matrizes e proporção entre o tamanho das *caches* e os conjuntos de trabalho de cada *benchmark*.

Benchmark	***Células*** **10 *threads***	**Bits**	***Cache* 256 %**	***Cache* 4K %**
MCNC20				
too-lrg	1.690	11	137,5%	110,0%
ex5p	7.290	13	162,5%	130,0%
apex4	9.000	14	175,0%	140,0%
alu4	9.610	14	175,0%	140,0%
misex3	10.240	14	175,0%	140,0%
apex2	11.560	14	175,0%	140,0%
ex1010	27.040	15	187,5%	150,0%
EPFL				
ctrl	640	10	125,0%	100,0%
int2float	810	10	125,0%	100,0%
router	1.210	11	137,5%	110,0%
cavlc	1.690	11	137,5%	110,0%
priority	2.560	12	150,0%	120,0%
adder	3.240	12	150,0%	120,0%
dec	3.610	12	150,0%	120,0%
i2c	4.000	12	150,0%	120,0%
bar	6.250	13	162,5%	130,0%
max	9.000	14	175,0%	140,0%
sin	16.810	15	187,5%	150,0%
voter	17.640	15	187,5%	150,0%
square	42.250	16	200,0%	160,0%
sqrt	51.840	16	200,0%	160,0%
multiplier	51.840	16	200,0%	160,0%
log2	82.810	17	212,5%	170,0%
div	104.040	17	212,5%	170,0%

A análise das Figuras 4.10 e 4.12 confirma que, conforme previsto, a utilização de

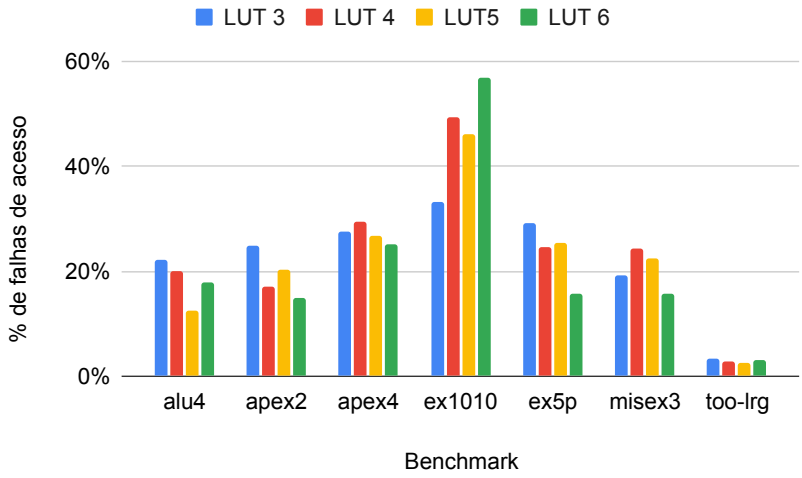


Figura 4.10: *Cache* 256 palavras. Total de falhas de acesso para os *benchmarks* MCNC20.

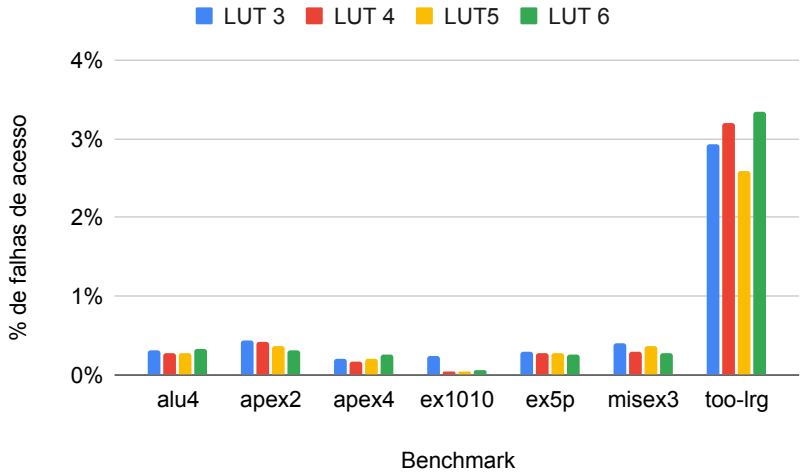


Figura 4.11: *Cache* 4096 palavras. Total de falhas de acesso para os *benchmarks* MCNC20.

uma *cache* de capacidade reduzida resulta em uma elevada taxa de falhas de acesso (*cache misses*). Entretanto, à medida que o tamanho da *cache* é ampliado, observa-se uma redução significativa na ocorrência dessas falhas. Especificamente, para a configuração com *cache* de 4096 palavras, a taxa de *misses* decresce para valores inferiores a 5%, conforme evidenciado nas Figuras 4.11 e 4.13 (a).

Ressalta-se, contudo, que o circuito *too-lrg* não apresentou variações significativas na taxa de falhas de leitura da *cache*. Esse comportamento pode ser atribuído às suas dimensões reduzidas, uma vez que circuitos de menor porte tendem a ser integralmente alocados na *cache*, como pode ser constatado na Tabela 4.12, o que contribui para a diminuição da taxa de falhas de acesso.

Ainda na Figura 4.13, em (b), observa-se uma atenuação dos ganhos em tempo de execução do algoritmo YOTO quando se emprega memória *cache*. Foram realizados

experimentos em quatro cenários distintos, nos quais a penalidade temporal associada a uma falha de *cache* foi parametrizada de forma crescente. Especificamente, os cenários com menor penalidade (caso w1) apresentaram tempos de recuperação mais curtos, enquanto aqueles com penalidade mais elevada (caso w8) implicaram tempos de recuperação substancialmente maiores para cada falha, conforme ilustrado na Figura 4.4.

Apesar do aumento progressivo no custo associado às falhas de *cache*, os resultados experimentais indicam que os ganhos de tempo de execução mantiveram-se, em grande parte, comparáveis aos obtidos na execução sem o uso de *cache*. Esse comportamento pode ser descrito pelo seguinte modelo analítico, o qual estima o tempo total de execução de uma *thread* ao incorporar explicitamente os efeitos das falhas de *cache*:

$$Total_{exec} = Exec + W * Cache_{falhas}$$

em que o termo $Total_{exec}$ denota o número total de ciclos consumidos durante a execução de uma *thread*, $Exec$ corresponde ao número de ciclos requeridos para uma execução ideal, isenta de falhas, W representa o custo associado a cada falha de *cache* e $Cache_{falhas}$ indica a quantidade total de falhas ocorridas. Essa formulação permite quantificar o impacto das falhas de *cache* sobre a latência total de execução.

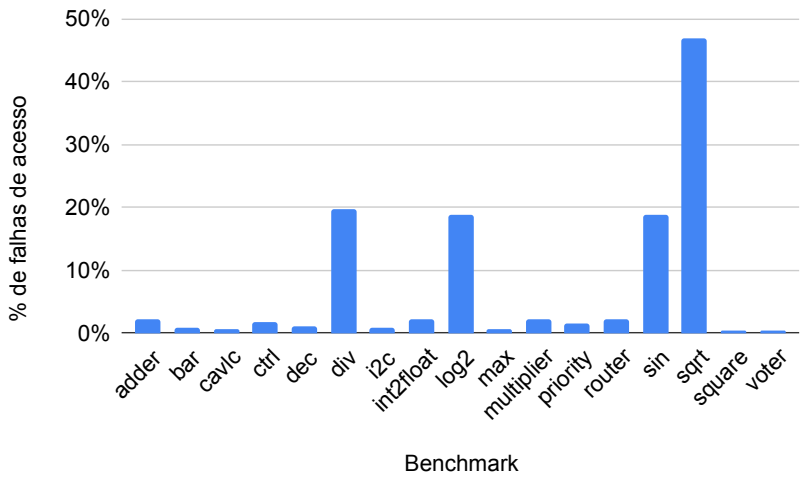


Figura 4.12: *Cache* 256 palavras. Total de falhas de acesso para os *benchmarks* EPFL. LUT 6

Esse comportamento associa-se às características do padrão de acesso do algoritmo YOTO, o qual tende a posicionar os nós do circuito em regiões adjacentes da matriz. Essa estratégia favorece elevada localidade espacial e temporal no acesso às células, o que contribui para o reaproveitamento de dados na *cache* e, como consequência, reduz a frequência de acessos diretos à memória principal (B-RAM).

Outro aspecto relevante refere-se ao custo do *hardware* associado à implementação

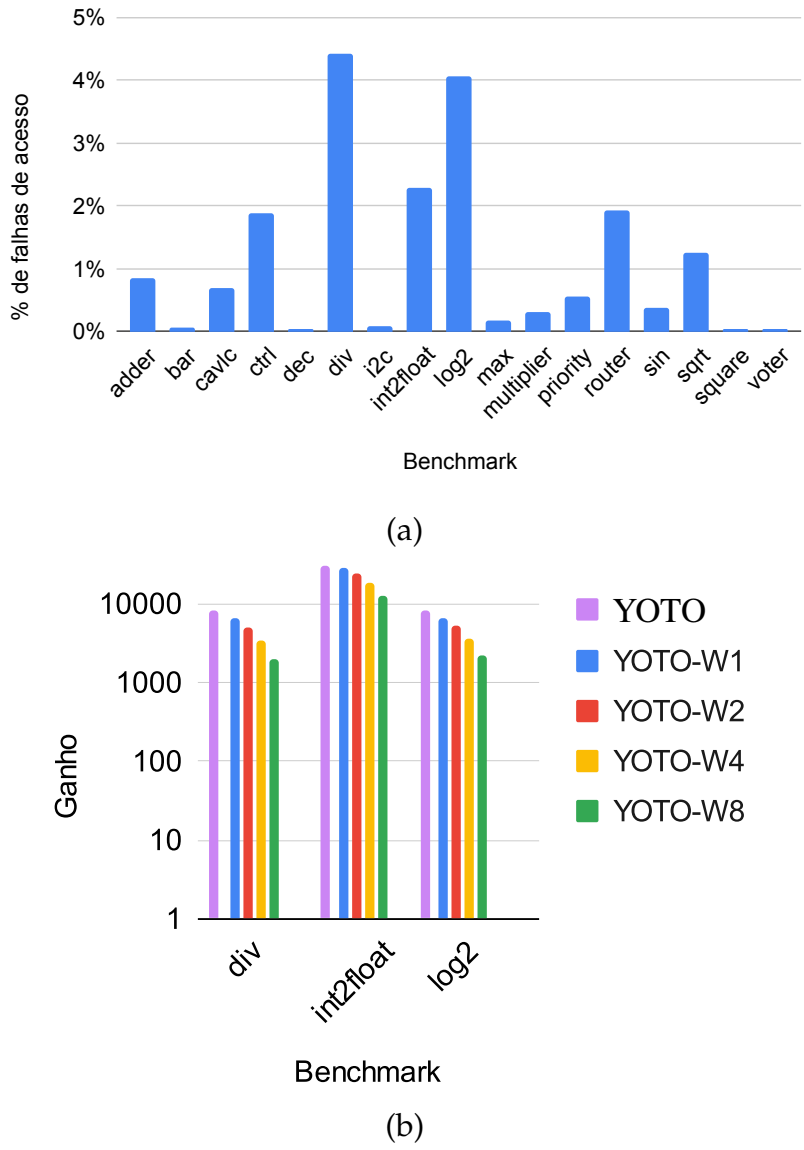


Figura 4.13: *Cache* 4096 palavras. Total de falhas de acesso para os *benchmarks* EPFL. LUT 6 (a). Efeito sobre o ganho nas execuções do YOTO (b).

da *cache*. Tomando-se o *benchmark div* como referência, observa-se que cada palavra armazenada na *cache* deve conter, no mínimo, 14 b, valor previamente identificado como necessário para o endereçamento das posições da matriz. Considerando-se que cada linha da *cache* é composta por 4 palavras, obtém-se um total de $4 \times 14 = 56$ b por linha, o que implica a utilização de 56 blocos de B-RAM para o armazenamento dos dados da *cache*.

Além disso, torna-se necessária a inclusão de, pelo menos, 1 b por linha para o controle de validade dos dados (bit de validade), o que acarreta a utilização de um bloco adicional de B-RAM. Para a identificação dos dados no cenário considerado, o sistema requer endereços de 14 b, suficientes para representar até 16 Mega posições de memória $(2^{14} = 16.384)$, bem como 4 b adicionais para a identificação da *thread*,

considerando a execução de 10 *threads*. Dessa forma, o endereçamento total demanda 18 b.

Assumindo-se uma *cache* com 1024 linhas (2^{10}), são necessários 10 b para o endereçamento interno das linhas da *cache*, além de 2 b para a identificação das 4 palavras por linha. Consequentemente, o campo *TAG* da *cache* deve armazenar os 6 b remanescentes $(18 - 10 - 2)$, o que requer a utilização adicional de 6 B-RAMs.

Portanto, o custo total estimado para a implementação dessa *cache* é de 56 B-RAMs para os dados, 1 para o controle de validade e 6 para o armazenamento do campo *TAG*, totalizando 63 blocos de B-RAM. Esse valor corresponde a aproximadamente 3% dos recursos disponíveis no dispositivo considerado. Em razão disso, a atualização da Tabela 3.1 é apresentada na Tabela 4.13. Esses resultados preliminares indicam que a utilização de uma *cache* de 4096 palavras produz um impacto reduzido na taxa de *cache misses*; contudo, a adoção de duas *caches* resulta em um aumento na utilização de recursos lógicos e de memória.

Tabela 4.13: Estimativa de uso de B-RAMs por algoritmo e *cache* de 4096 palavras para o maior *benchmark*.

Algoritmo	**C2N**	**N2C**	**Total Parcial (%)**	**Viz**	**Arestas**	**Anot.**	**Total (%)**
SA	56	224	280 (13%)	56	-	-	336 (15%)
YOTO	56	56	112 (5%)	-	28	-	140 (7%)
YOTT	56	112	168 (8%)	-	28	66	262 (12%)
CACHE (B-RAM 1b)	66	66	132 (6%)				
CACHE (B-RAM 8b)	9	9	18 (1,5%)				
CACHE (URAM 72b)	1	1	2 (0,1%)%				

De acordo com a documentação técnica da Xilinx [4], os blocos de memória RAM embarcada (B-RAM) disponíveis em dispositivos FPGA da família Versal permitem configurações flexíveis de largura de dados, o que viabiliza acessos com diferentes tamanhos de palavra. Ao se empregar palavras de 8 bits, obtêm-se blocos com capacidade de 4 K. Considerando a necessidade estimada de 66 bits por instância para a implementação proposta, esses dados podem ser organizados em palavras de 8 bits, o que resulta em $\frac{66}{8} \approx 9$ blocos para a implementação de cada *cache* associada às estruturas N_2C e C_2N. Essa configuração reduz de forma significativa o consumo global de B-RAMs, correspondendo a aproximadamente 0,7% do total disponível no dispositivo analisado.

Como terceira alternativa para a implementação das memórias *cache*, pode-se considerar o emprego de memórias URAMs. Segundo a documentação oficial da Xilinx, o dispositivo FPGA VU9P, utilizado como referência nos exemplos apresentados neste capítulo, disponibiliza um total de 960 blocos de URAM, resultando em uma

capacidade agregada de 270 Mb, com 288 Kb por bloco [3]. Cada bloco de URAM é configurado como uma memória organizada em 4 K linhas, com largura de palavra de 72 bits [2].

Dessa forma, ao se considerar a implementação de uma memória *cache* composta por 1 K linhas, cada uma com 66 bits de largura, verifica-se que um único bloco de URAM é suficiente para acomodar toda a estrutura. Essa característica representa uma vantagem expressiva em relação ao uso de blocos de B-RAM, sobretudo em projetos de grande escala, nos quais a pressão sobre os recursos de memória embutida é elevada.

Entretanto, a escolha do tamanho da *cache* deve considerar não apenas os requisitos funcionais do algoritmo, mas também a granularidade de alocação imposta pelos blocos físicos de memória disponíveis no dispositivo. Um dimensionamento inadequado pode resultar em subutilização da capacidade disponível ou no desperdício de recursos de memória, o que compromete a eficiência global da implementação.

Na Tabela 4.13, os valores correspondentes às estruturas **Viz**, **Arestas** e **Anot.** foram omitidos nas linhas que apresentam os resultados referentes ao uso de *cache*. Essa decisão decorre do fato de que tais estruturas não se encontram presentes de forma padronizada em todos os circuitos empregados nos experimentos, o que inviabiliza uma comparação homogênea entre as diferentes configurações analisadas.

Dessa forma, optou-se por avaliar o impacto do uso de *cache* exclusivamente nas estruturas C_2N e N_2C, de modo a permitir uma comparação direta entre as versões com e sem acesso intermediado por *cache*. Deve-se considerar, adicionalmente, que, em razão de o algoritmo YOTT replicar a memória N_2C, torna-se necessária uma instância adicional de memória *cache* para esse acesso. Consequentemente, o total de B-RAMs requerido para sua utilização deve ser duplicado em relação aos valores apresentados na Tabela.

Em síntese, a implementação com B-RAMs de 1 bit demandaria 198 blocos, enquanto a utilização de B-RAMs de 8 bits exigiria 27 blocos. Alternativamente, a adoção de URAMs com largura de 72 bits permitiria a implementação com apenas 3 blocos.

4.2.5 Análise Estrutural dos Grafos e Impacto no Posicionamento

À medida que a pesquisa evoluiu, o foco concentrou-se no algoritmo YOTO, por apresentar, dentre os três algoritmos propostos, o maior ganho em tempo de execução aliado a um comprimento de caminho crítico considerado aceitável em relação à sua velocidade. Também adotou-se a versão 9 do VPR, que apresenta maior nível de otimização e menor tempo de execução quando comparada à versão anterior. Com isso, observou-se que, o ganho de desempenho em tempo de execução entre os algoritmos desta tese e o VPR reduziu aproximadamente uma ordem de grandeza com a utilização da versão mais recente da ferramenta.

Esse comportamento motivou uma investigação acerca das causas da redução do ganho relativo do YOTO. A análise dos relatórios de execução indicou que, para o *benchmark Mnist 1*, utilizado como estudo de caso, o circuito contém aproximadamente quatro mil LUTs a serem posicionadas. Observou-se que o algoritmo necessitava, em média, de cerca de 100 tentativas para encontrar uma célula livre adequada para o posicionamento de um único nó na matriz.

A Figura 4.14 apresenta os *boxplots* dos instantâneos do posicionamento incremental utilizando a estratégia convencional de busca em largura para o *benchmark* MNIST 1, composto por 4.102 LUTs mapeadas em uma matriz de 65×65, com taxa de ocupação elevada, equivalente a 97% das LUTs. O posicionamento final registra número médio de tentativas por nó próximo de 100, com mediana igual a 5.

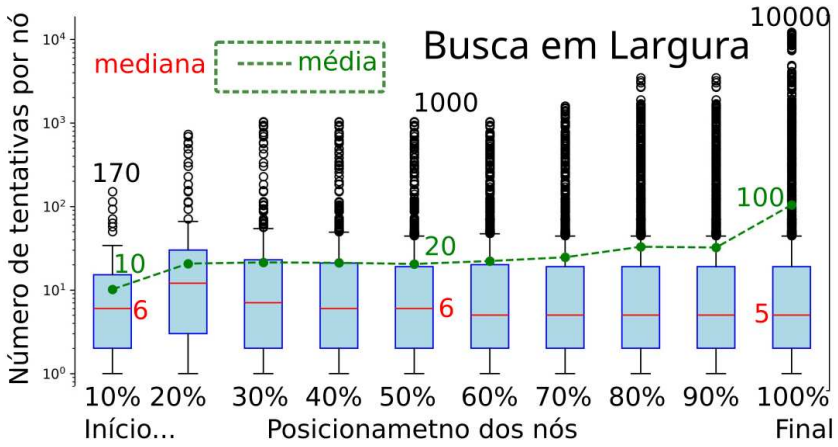


Figura 4.14: Mnist 1 - Posicionamento com busca em largura dividido em 10 partes cumulativas.

Diante desse cenário, realizou-se um estudo comparativo das características estruturais dos grafos pertencentes aos três domínios considerados. A Tabela 4.14 apresenta as características de dois *benchmarks* de cada domínio com exemplos de menor e maior porte.

A comparação entre os *benchmarks* dos domínios FPGA, CGRA e QCA mostra diferenças estruturais relevantes que influenciam o comportamento dos algoritmos de posicionamento. Nos grafos associados a FPGAs, observa-se uma maior densidade de conexões, com grau médio superior a 4 em alguns casos, além de uma profundidade significativa. O circuito *div*, por exemplo, apresenta mais de 40 mil arestas e comprimento de caminho mais longo igual a 859 níveis, o que caracteriza um forte encadeamento estrutural e elevada concentração de dependências.

A Figura 4.15 mostra um posicionamento obtido para o *benchmark* MNIST 1. A representação em mapa de calor indica a quantidade de tentativas necessárias para o posicionamento de cada LUT. Observa-se que a maioria dos elementos necessita de um

Tabela 4.14: Características estruturais dos *benchmarks* avaliados.

Benchmark	Nós	Arestas	Grau Médio	Caminho Mais Longo
FPGA				
too-lrg	144	479	3,33	6
ex1010	2462	11319	4,60	7
ctrl	59	154	2,61	3
div	10139	40840	4,03	859
QCA				
xor5R_inverted	38	48	1,26	12
CGRA				
cosine2	83	93	1,12	7
Fir16x8	616	728	1,18	22

número reduzido de tentativas, o que explica a mediana apresentada na Figura 4.14. Entretanto, um conjunto restrito de casos exige número maior de tentativas, o que eleva a média total.

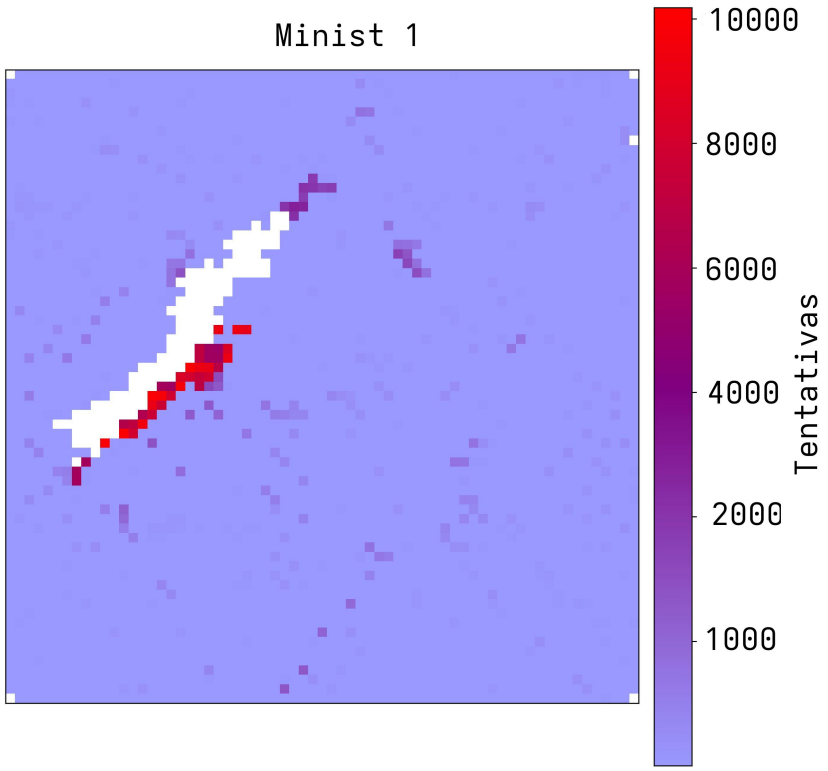


Figura 4.15: Mnist 1 - Mapa de calor da quantidade de tentativas para o posicionamento de cada LUT.

Essa configuração impõe pressão sobre regiões da matriz durante o posiciona-

mento, sobretudo nas proximidades de nós pertencentes ao caminho crítico. Como consequência, o posicionador enfrenta elevada competição por células adjacentes disponíveis.

Em contraste, os grafos utilizados nos domínios CGRA e QCA apresentam um grau médio substancialmente inferior, tipicamente próximo de 1,2, o que indica uma estrutura mais esparsa e regular. A limitação de *fan-in* e *fan-out* nesses domínios contribui para reduzir a densidade de conexões e, consequentemente, a pressão estrutural sobre a matriz de posicionamento.

O contraste estrutural descrito explica o comportamento observado no algoritmo YOTO. Em circuitos densos e profundos, como o *div*, a natureza gulosa do algoritmo tende a concentrar alocações em regiões de alta conexão, o que satura antecipadamente de áreas da matriz e aumenta o número de tentativas necessárias para a localização de células livres. Em grafos mais esparsos, como os associados a CGRA, a dispersão estrutural reduz a ocorrência de bloqueios locais e preserva o desempenho do algoritmo. Com essas observações, propôs-se as estratégias vistas na Seção 3.3.5 e os resultados podem ser observados na próxima Seção.

4.2.6 Resultados da Estratégia Híbrida

A Figura 4.16 ilustra o impacto da aplicação da uma estratégia híbrida de busca em largura e profundidade, com limiar de 12 tentativas, associada à busca por LUTs livres por varredura quando a ocupação atinge entre 80% e 90% propostas na Seção 3.3.5. Nessa configuração, o valor médio de tentativas reduz-se para 9, com mediana igual a 3. Consequentemente, o tempo de execução, proporcional ao número de tentativas multiplicado pelo número de nós, reduz-se em uma ordem de grandeza. Observa-se, adicionalmente, redução dos casos extremos, o que impacta diretamente tanto a média quanto o tempo final de execução.

Nos experimentos realizados, as comparações foram feitas tendo o VPR v9 como referência [29]. Adotou-se como linha de base o método de posicionamento por *bounding box* do VPR v9, com os parâmetros padrão *–inner_num=0.5* e *–place_algorithm=bounding_box*, cujo objetivo consiste na minimização do comprimento total dos fios do circuito, sendo essa configuração identificada como **VPR BB**. Adicionalmente, avaliou-se a opção *criticality_timing*, que busca reduzir simultaneamente o comprimento dos fios e o atraso das conexões, denominada **VPR CT**.

Para os posicionamentos gerados pelo YOTO, foram avaliados quatro cenários, denominados Guloso_6, BuscaL_6, Guloso_60 e BuscaL_60. As configurações denominadas Guloso_x utilizam busca em largura no posicionamento local, sendo que a primeira executa seis instâncias paralelas de posicionamento, enquanto a segunda executa 60 instâncias. O custo de execução de uma ou de seis instâncias é equivalente,

Tabela 4.15: Resultados de posicionamento para o VPR v9 e o YOTO: comparação entre o tempo de execução e o caminho crítico mapeado.

		VPR 9.0		Guloso_6			BuscaL_6		Guloso_60		BuscaL_60	
Benchmark	**Métricas**	**BB**	**CT**	***Software***	**manh**	**slack**	**manh**	**slack**	**manh**	**slack**	**manh**	**slack**
mnist1	crítico	1,00	0,93		1,34	1,31	1,81	1,53	1,47	1,23	1,65	1,66
	execução	38,42 s	53,0 s	164,54 ms	15,69 ms		1,85 ms		185,21 ms		18,27 ms	
4.102 LUTs	ganho	1,0	0,7	234	2.448		20.670		207		2.102	
mnist2	crítico	1,0	0,9		1,26	1,19	1,51	1,42	1,21	1,19	1,31	1,4
	execução	44,89 s	61 s	76,5 ms	10,57 ms		2,72 ms		98,08 ms		26,03 ms	
4.421 LUTs	ganho	1,0	0,74	587	4.243		16.486		458		1.724	
mnist3	crítico	1,0	0,92		1,5	1,63	1,58	1,64	1,74	1,49	1,6	1,58
	execução	70,97 s	100,19 s	109,3 ms	19,06 ms		2,45 ms		283,73 ms		25,43 ms	
5.990 LUTs	ganho	1,0	0,7	649	3.722		28.876		250		2.790	
mnist4	crítico	1,0	0,98		1,42	1,52	1,68	1,84	1,44	1,4	1,63	1,74
	execução	77,88 s	110,32 s	109,83 ms	17,01 ms		3,34 ms		169,63 ms		30,09 ms	
6.385 LUTs	ganho	1,0	0,7	709	4.576		23.276		459		2.588	
jsc1	crítico	1,0	1,04		1,06	1,14	1,32	1,2	1,22	1,1	1,22	1,37
	execução	1,91 s	2,68 s	11,52 ms	999,64 μs		137,28 μs		12,34 ms		1,75 ms	
940 LUTs	ganho	1,0	0,71	166	1.911		12.594		155		1.090	
jsc2	crítico	1,0	0,98		1,42	1,53	1,46	1,55	1,36	1,55	1,58	1,63
	execução	11,96 s s	16,65 s	34,56 ms	4,585 ms		813,31 μs		40,57 ms		7,37 ms	
2.686 LUTs	ganho	1,0	0,72	346	2.608		14.705		295		1.621	
max	crítico	1,0	0,91		0,92	0,84	0,9	0,87	0,91	0,89	0,92	0,89
	execução	2,69 s	3,46 s	13,28 ms	3,07 ms		692,35 μs		26,12 ms		6,05 ms	
763 LUTs	ganho	1,0	0,78	204	873		3.885		103		444	
sin	crítico	1,0	1,03		1,26	1,22	1,26	1,24	1,36	1,18	1,32	1,19
	execução	4,51 s	5,94 s	32,65	6,7 ms		699,07 μs		56,99 ms		9,07 ms	
1.475 LUTs	ganho	1,0	0,76	138	673		6.451		80		497	
voter	crítico	1,0	1,05		1,63	1,58	1,57	1,63	1,49	1,45	1,59	1,72
	execução	10,52 s	11,19 s	15,43 ms	2,77 ms		579,96 μs		26,13 ms		5,41 ms	
1.548 LUTs	ganho	1,0	0,94	681	3.787		18.139		402		1.941	
square	crítico	1,0	0,76		0,71	0,66	0,74	0,73	0,66	0,77	0,73	0,78
	execução	20,89 s	31,57 s	249,23	46,76 ms		5,58 ms		474,91 ms		81,26 ms	
3.933 LUTs	ganho	1,0	0,66	84	447		3.744		44		257	
sqrt	crítico	1,0	0,98		1,0	0,96	1,07	1,11	0,96	0,92	1,08	1,09
	execução	31,8 s	46,64 s	161,44	26,05 ms		4,34 ms		159,06 ms		126,88 ms	
4.784 LUTs	ganho	1,0	0,68	197	1.220		7.314		200		2.507	
log2	crítico	1,0	0,94		1,57	1,55	1,68	1,73	1,52	1,48	1,6	1,6
	execução	77,34 s	116,8 s	405,99	78,08 ms		22,39 ms		556,4 ms		306,42 ms	
7.906 LUTs	ganho	1,0	0,66	191	990		3.454		139		252	
div	crítico	1,0	0,99		0,89	0,86	0,92	0,93	0,91	0,87	0,93	0,9
	execução	104,84 s	149,87 s	930,98	164,1 ms		51,16 ms		1,6 s		240,41 ms	
9.883 LUTs	ganho	1,0	0,7	113	639		2.048		65		436	
AVG	crítico		0,95		1,26		1,38		1,27		1,37	
	ganho		0,73	359	2.541		17.751		322		1.837	

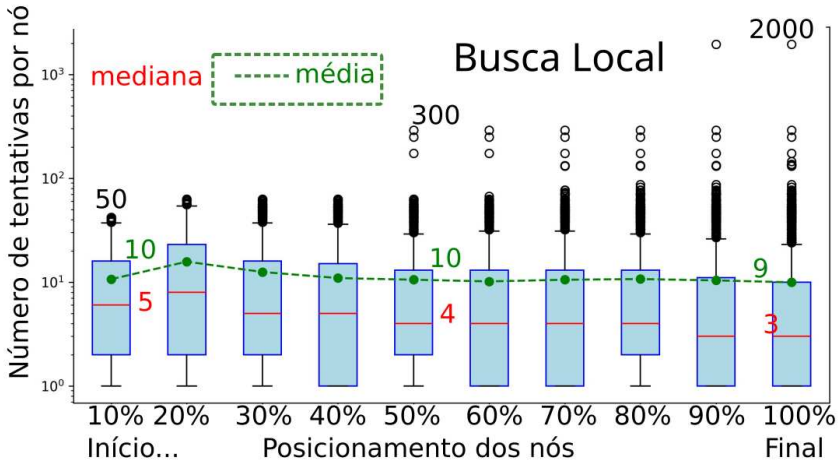


Figura 4.16: Mnist 1 - Posicionamento híbrido dividido em 10 partes cumulativas.

uma vez que a unidade de *hardware* possui seis estágios de *pipeline*. Também foi considerada uma implementação gulosa em *software*, executada em uma CPU. A unidade de *hardware* opera a uma frequência de 250 MHz em FPGA. Assim, embora o *clock* do *hardware* seja aproximadamente dez vezes inferior ao da CPU, o tempo médio de execução mostrou-se 7,08× menor que o da solução em *software* para o Guloso_6 e 49,4× menor para o BuscaL_6.

Foram avaliadas duas heurísticas simples para a seleção do melhor posicionamento entre múltiplas instâncias. A primeira baseada no contador de distância de Manhattan mínima, que acumula a soma das distâncias de todas as arestas durante o posicionamento, sendo identificada como **manh** na Tabela 4.15. O posicionamento associado ao menor valor final desse contador é selecionado para a etapa de roteamento.

A segunda heurística utiliza o conceito de folga temporal (*slack*). Para cada aresta da representação do grafo de entrada, adicionou-se um campo adicional destinado ao armazenamento da folga temporal, definida como a diferença entre os tempos *as-soon-as-possible* (ASAP) e *as-late-as-possible* (ALAP). Uma aresta pertence ao caminho crítico quando apresenta folga igual a zero, enquanto valores positivos indicam arestas fora do caminho crítico. Durante o posicionamento, soma-se ao contador global, para cada aresta, a distância de Manhattan menos o valor da folga temporal. Ao final do processo, seleciona-se a instância que apresenta o menor valor total acumulado, considerando-se seis ou 60 instâncias paralelas, conforme o experimento.

Além das seis implementações avaliadas para os conjuntos de dados MNIST e JSC, também foram considerados circuitos aritméticos do conjunto de *benchmarks* EPFL, com tamanhos variando entre 1.000 e 10.000 LUTs. Em todos os experimentos, utilizou-se o fluxo Yosys/ABC para o mapeamento em LUTs de seis entradas com ojá explicado anteriormente.

A Tabela 4.15 encontra-se organizada em grupos de três linhas por circuito. A

primeira linha, identificada como *critico*, apresenta o caminho crítico após o roteamento do VPR v9, normalizado em relação ao posicionamento por *bounding box* (coluna **BB**, cujo valor de referência é sempre igual a um). A coluna **VPR CT** indica a melhoria no atraso crítico obtida com o uso do posicionamento orientado por criticidade temporal, que apresenta, em média, um ganho de aproximadamente 5% em relação ao **VPR BB**. Contudo, essa melhoria ocorre à custa de um aumento no tempo de execução do posicionamento, resultando em uma desaceleração média de 1,37×.

O tempo de execução do posicionamento no VPR v9 variou entre 2 segundos e 2 minutos, o que inviabiliza sua aplicação em cenários de execução em tempo real. Em contraste, o tempo de execução do posicionamento com a unidade de *hardware* proposta varia entre 1 e 76 milissegundos para o método guloso, e entre 131 μs e 51 milissegundos para a estratégia aprimorada de busca local. Dessa forma, a unidade de posicionamento em *hardware* atinge ganhos de desempenho de três a quatro ordens de magnitude em relação ao VPR v9. Em três circuitos (*max*, *square* e *div*), o posicionamento em *hardware* resultou, inclusive, na redução do caminho crítico. Em média, os métodos Guloso_6 e BuscaL_6 aumentaram o caminho crítico em 26% e 38% em relação ao **VPR BB**, respectivamente, ao mesmo tempo em que alcançaram ganhos médios de velocidade de 2.541× e 17.751×.

Também foi avaliada a execução de um número dez vezes maior de instâncias paralelas, por meio das configurações Guloso_60 e BuscaL_60. Contudo, como apenas duas instâncias foram consideradas na etapa de roteamento, os ganhos observados foram pequenos. Esses resultados indicam a possibilidade de exploração mais aprofundada do espaço de soluções, com o desenvolvimento de critérios adicionais de seleção que possam ser implementados diretamente em *hardware*, de forma semelhante ao contador proposto, permitindo a escolha dinâmica da melhor solução durante a exploração paralela de múltiplas instâncias.

4.3 SA, YOTO e YOTT como posicionadores para QCAs

Para uma avaliação preliminar da proposta de posicionamento em QCAs, foram realizados experimentos iniciais com a utilização do algoritmo de SA sobre um conjunto de *benchmarks* disponibilizados em [78]. Os testes foram executados em um sistema computacional equipado com processador Intel Xeon E5-2630 v3 @ 2,40 GHz, 64 GB de memória RAM DDR4 a 2133 MHz e sistema operacional Linux Ubuntu 24.04 LTS. A implementação do algoritmo foi desenvolvida na linguagem *C++*.

Para fins de comparação dos resultados obtidos, foram adotados como referência os valores reportados na Tabela 1 apresentada em [78]. A preparação dos *benchmarks* envolveu etapas de conversão e processamento dos circuitos originalmente descritos em Verilog. Inicialmente, os circuitos foram sintetizados com o auxílio da ferramenta

ABC [60], de modo a gerar representações no formato BLIF, configuradas para portas de duas entradas. Essa configuração foi adotada em função das restrições impostas pela topologia QCA considerada, USE, a qual admite duas entradas e duas saídas por célula.

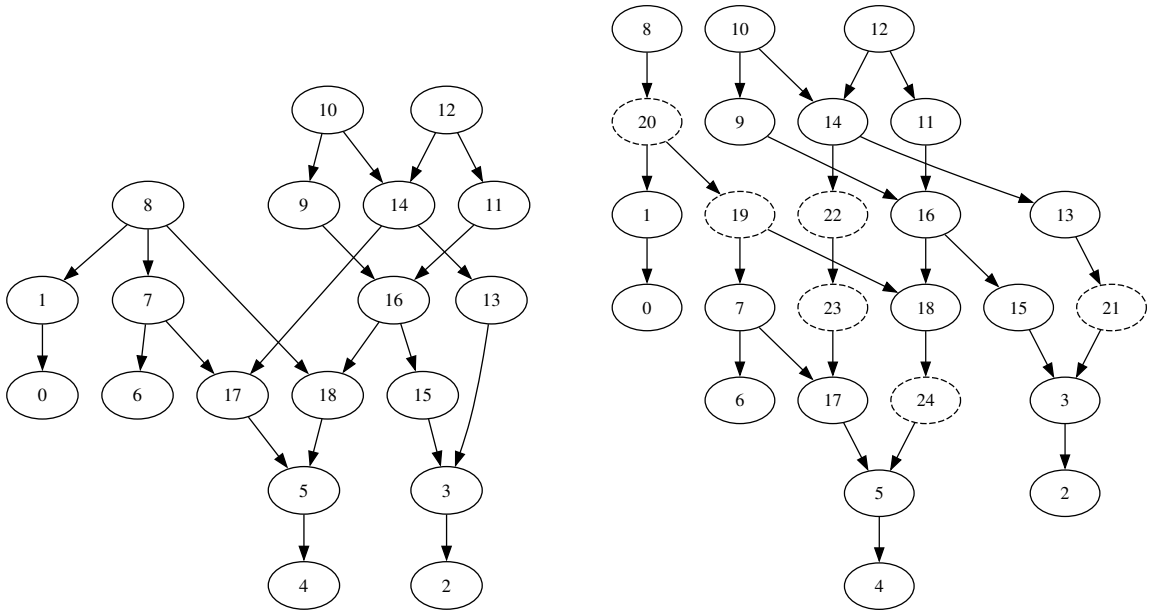
A partir dos arquivos BLIF gerados, os circuitos foram convertidos para o formato DOT, utilizado tanto para fins de visualização quanto como fonte de dados para os algoritmos de posicionamento. Em seguida, os grafos resultantes foram submetidos a um processo de balanceamento, com o objetivo de assegurar que todos os caminhos de propagação de sinal até as saídas apresentassem profundidade equivalente. Essa etapa é essencial para garantir o funcionamento correto da topologia USE, uma vez que a diferença no tempo de chegada dos sinais compromete o comportamento funcional do circuito.

Posteriormente, foi incluída uma etapa específica de correção de *fan-out*, com o objetivo de assegurar que cada nó apresentasse, no máximo, duas conexões de saída, em conformidade com as restrições impostas pela topologia USE. Essa exigência decorre do fato de que, durante o processo de síntese conduzido pela ferramenta ABC, direcionado à implementação em portas de duas entradas, não são consideradas limitações relativas ao número de conexões de saída das portas lógicas. Em particular, embora o ABC respeite a restrição de *fan-in*, a ferramenta não impõe mecanismos de controle sobre o *fan-out*.

Dessa forma, o posicionamento e o roteamento são tratados de maneira integrada. A inserção dos nós do tipo *fio* foi feita de modo a adequar o grafo às restrições de fan-in e fan-out impostas pela topologia USE, além de balancear as conexões. Como consequência, uma vez que todos os vértices do grafo sejam posicionados em células válidas e em conformidade com o sentido de propagação do sinal, o circuito resultante estará também roteado. Dessa forma, um posicionamento correto implica a resolução automática do roteamento, diferentemente do que ocorre em arquiteturas FPGA e CGRA, nas quais o posicionamento e o roteamento constituem etapas distintas.

A Figura 4.17 ilustra o resultado da execução com o circuito *B1_r2*. Na Figura 4.17(a), apresenta-se o grafo obtido a partir da conversão realizada pela ferramenta ABC e da posterior transformação para o formato DOT. Na Figura 4.17(b), é exibido o grafo ajustado após a incorporação do balanceamento dos caminhos e a inserção de nós auxiliares destinados à correção do *fan-out*.

Como exemplo, observa-se na Figura 4.17(a) a conexão direta entre os nós 14 e 17. Na versão ajustada apresentada na Figura 4.17(b), essa conexão passa a ser mediada pelos nós auxiliares 22 e 23. De forma análoga, a saída do nó 8, originalmente conectada diretamente aos nós 1, 7 e 18 em (a), é redistribuída em (b) por meio dos nós auxiliares 19 e 20. Os nós representados por linhas tracejadas identificam precisamente esses elementos intermediários inseridos no processo de adequação do grafo.



(a) *Benchmark B1_r2* antes do balanceamento.

(b) *Benchmark B1_r2* após o balanceamento.

Figura 4.17: *Benchmark B1_r2* antes (a) e depois (b) do balanceamento e correção de *fan-out*.

Nos experimentos iniciais, não foram consideradas as funcionalidades de cruzamento de fios e de derivação de sinais no interior de uma mesma célula QCA. Como consequência, cada célula da matriz admite apenas um único elemento funcional, seja ele um fio, uma porta lógica ou um pino de entrada ou saída. Essa restrição impõe maior rigidez ao processo de alocação dos componentes e interfere diretamente na densidade de ocupação do circuito, o que aumenta a complexidade do problema de posicionamento. Além disso, tal limitação pode inviabilizar o mapeamento de circuitos cujos grafos não sejam planares, uma vez que, na ausência de cruzamentos, determinadas conexões tornam-se complexas. Uma alternativa para contornar essa limitação consiste em permitir a alocação simultânea de dois nós do tipo *fio* em uma mesma célula, o que viabiliza a implementação de cruzamentos.

A incorporação futura das funcionalidades de cruzamento de fios permitirá a implementação de múltiplas trilhas de sinal em uma mesma célula, seja por meio de direções mutuamente ortogonais, seja por meio da disposição paralela de sinais ao longo de um mesmo eixo, conforme permitido pela topologia USE. De forma complementar, a funcionalidade de derivação de sinais possibilitará que um único nó de entrada propague seu valor diretamente para múltiplos nós de saída, eliminando a necessidade de nós intermediários e reduzindo a complexidade do grafo a ser posicionado.

A Tabela 4.16 apresenta os resultados preliminares obtidos com a aplicação do

algoritmo de SA proposto, em comparação com os resultados obtidos pelo algoritmo heurístico descrito em [78]. Os dados são organizados de modo que se possa observar três métricas principais: a área total da matriz de posicionamento (**Área**), o número de células efetivamente ocupadas por componentes lógicos (**Ocupação**) e a diferença entre as áreas (**Diferença**).

Tabela 4.16: Resultados preliminares para área de posicionamento.

Bench	***Portas***	**SA proposto**			***Heuristic Algorithm ****		**Diferença**
		Área	**Ocup.**	**Arestas Subót. (%)**	**Área**	**Ocup.**	
FA	10	25	17 (68,0%)	1 (05%)	56	36 (64,2%)	31
B1_r2	19	42	25 (59,5%)	3 (10%)	60	44 (73,3%)	18
t	17	42	22 (52,3%)	1 (04%)	49	38 (77,5%)	7
newtag	24	64	32 (50,0%)	2 (05%)	80	44 (55,0%)	16
XOR5_r1	32	100	54 (54,0%)	4 (06%)	195	97 (49,7%)	95
CLPL	26	110	71 (64,5%)	3 (04%)	132	64 (48,4%)	22
XOR5_r	38	121	75 (61,9%)	11 (12%)	252	252 (100%)	131

Ao analisar os resultados apresentados na Tabela 4.16, observa-se que o algoritmo SA proposto apresenta maior eficiência no que se refere à área total necessária para o posicionamento dos circuitos. Em todos os *benchmarks* avaliados, a área da matriz obtida pelo SA é inferior àquela resultante do algoritmo heurístico de referência, com exceção do circuito *t*, no qual a área obtida pelo método proposto é 42, frente a 49 do algoritmo da literatura. Esses resultados indicam a capacidade do algoritmo proposto de produzir soluções mais compactas e reforçam a adequação da função de custo adotada.

A métrica de **ocupação** deve ser interpretada de forma conjunta com a área total da matriz. Por exemplo, no circuito *FA*, a ocupação de 17 células em uma matriz de área 25 corresponde a uma ocupação de 68%. Valores elevados de ocupação indicam menor quantidade de células livres, o que tende a dificultar o posicionamento especialmente no final, no qual a busca por posições válidas torna-se mais restrita. Observa-se que o algoritmo SA proposto atinge taxas de ocupação superiores a 60% em três casos, como nos circuitos *FA*, *CLPL* e *XOR5_r*.

Entretanto, uma taxa elevada de ocupação não implica, necessariamente, uma solução superior. Em QCAs, uma boa parte da área ocupada corresponde a nós auxiliares, tais como fios, e não a portas lógicas propriamente ditas. Essa característica traz a necessidade da análise conjunta do número de portas lógicas de cada circuito, apresentado na coluna **Portas** da tabela. Em vários *benchmarks*, observa-se que o número de portas lógicas representa apenas uma fração do total de células ocupadas. Isso indica que a compactação espacial decorre, em grande medida, da organização

dos fios para o balanceamento do circuito posicionado.

Essa relação explica casos em que o algoritmo heurístico de referência apresenta taxas de ocupação elevadas, como no circuito *t* (77,5%), porém à custa de uma matriz significativamente maior. De modo semelhante, no circuito *XOR5_r*, o método de referência ocupa integralmente a matriz (100%), evidenciando a ausência de flexibilidade para acomodação incremental e adaptação do posicionamento.

De forma geral, os resultados indicam uma relação equilibrada entre a compactação da área e a eficiência de utilização, no qual o algoritmo SA proposto apresenta desempenho favorável. Ressalta-se, contudo, que os resultados possuem caráter preliminar e não consideram, nesta etapa, funcionalidades como cruzamento e derivação de fios. Além disso, nenhum dos *benchmarks* foi completamente posicionado com custo zero, uma vez que, em todos os casos, ao menos uma aresta permanece em condição subótima. A quantidade de arestas nessa condição encontra-se na coluna *Arestas Subót.* da Tabela 4.16. Espera-se que tais limitações sejam mitigadas em trabalhos futuros por meio das extensões previstas para o modelo.

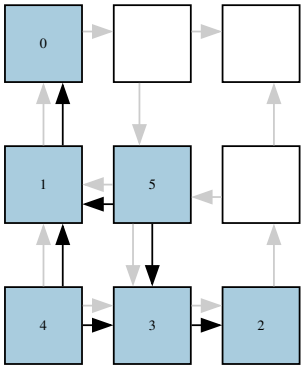
A Figura 4.18 apresenta quatro configurações distintas de posicionamento para o *benchmark Half Adder*, cada uma caracterizada por um número crescente de camadas adicionais de interconexões, variando de zero a três.

A princípio, a inclusão de camadas adicionais de fios tende a resultar em uma maior área ocupada, uma vez que tal estratégia implica o acréscimo de nós na matriz de posicionamento. Contudo, os resultados obtidos demonstram que essa relação não é necessariamente proporcional. Ao comparar os casos (a) e (b), verifica-se que a introdução de uma camada adicional de fios em (b) levou à redução da área total da matriz, a qual passou de nove para oito células. Essa redução decorre do fato de que a maior flexibilidade proporcionada pela camada adicional permitiu ao algoritmo identificar uma solução mais compacta para o problema de posicionamento.

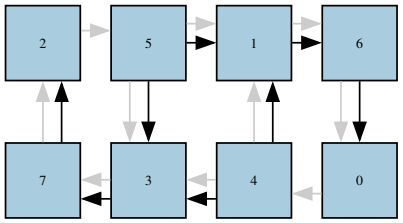
Comportamento análogo é observado nos casos (c) e (d), correspondentes, respectivamente, às configurações com duas e três camadas adicionais de fios. Embora o caso (d) necessite do posicionamento de um maior número de nós, a solução final demanda 24 células, enquanto a configuração com duas camadas adicionais ocupa uma área superior, totalizando 25 células.

Esses resultados indicam que a relação entre o número de camadas de fios disponíveis e a área final do circuito apresenta um comportamento não linear, condicionado pela topologia do grafo e pelas restrições de roteamento oferecidas pelo modelo de posicionamento adotado. Essa flexibilidade adicional indica que o papel das camadas extras de fios é um recurso de otimização, ao mesmo tempo em que traz o desafio de determinar a quantidade ideal dessas camadas e sua distribuição espacial no circuito.

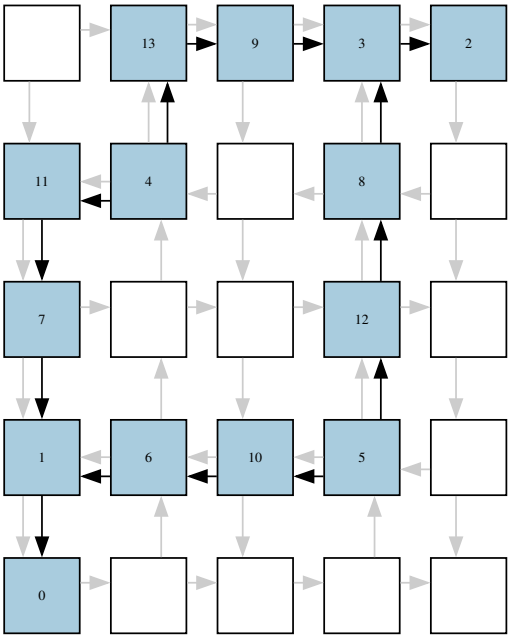
Outra observação importante, identificada a partir da análise dos posicionamentos gerados pelo algoritmo SA, refere-se à ausência de critérios explícitos de compactação



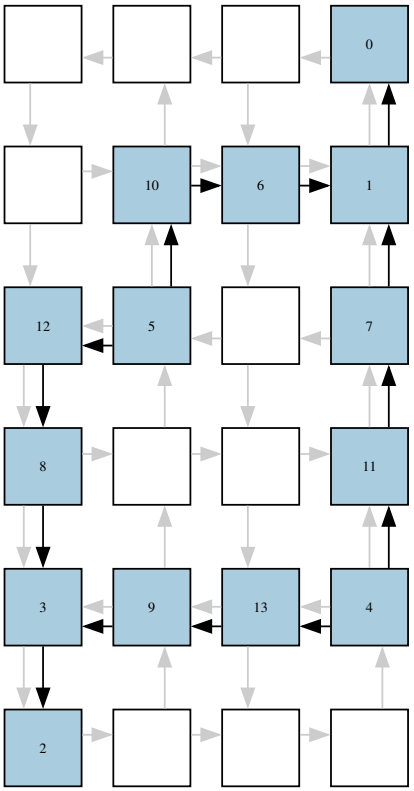
(a) *Benchmark Half Adder*.



(b) *Benchmark Half Adder* com uma camada extra.



(c) *Benchmark Half Adder* com duas camadas extras.



(d) *Benchmark Half Adder* com três camadas extras.

Figura 4.18: Posicionamentos obtidos pelo SA para o *benchmark Half Adder* de 0 a 3 camadas de fios extras.

na função de custo atualmente adotada. Embora o algoritmo apresente uma tendência à geração de soluções mais densas, a função de custo não incorpora termos de penalização ou de incentivo associados à área total ocupada na matriz de posicionamento. Como consequência, decisões subótimas podem ser adotadas ao longo do processo de busca, resultando em arranjos que, embora viáveis do ponto de vista funcional, não correspondem necessariamente às soluções mais eficientes em termos de utilização do

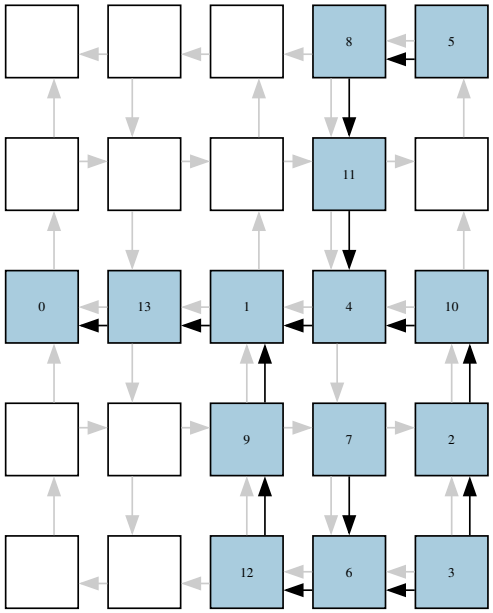


Figura 4.19: Uso extra de área de posicionamento pelo SA para o *benchmark 2:1 Mux*.

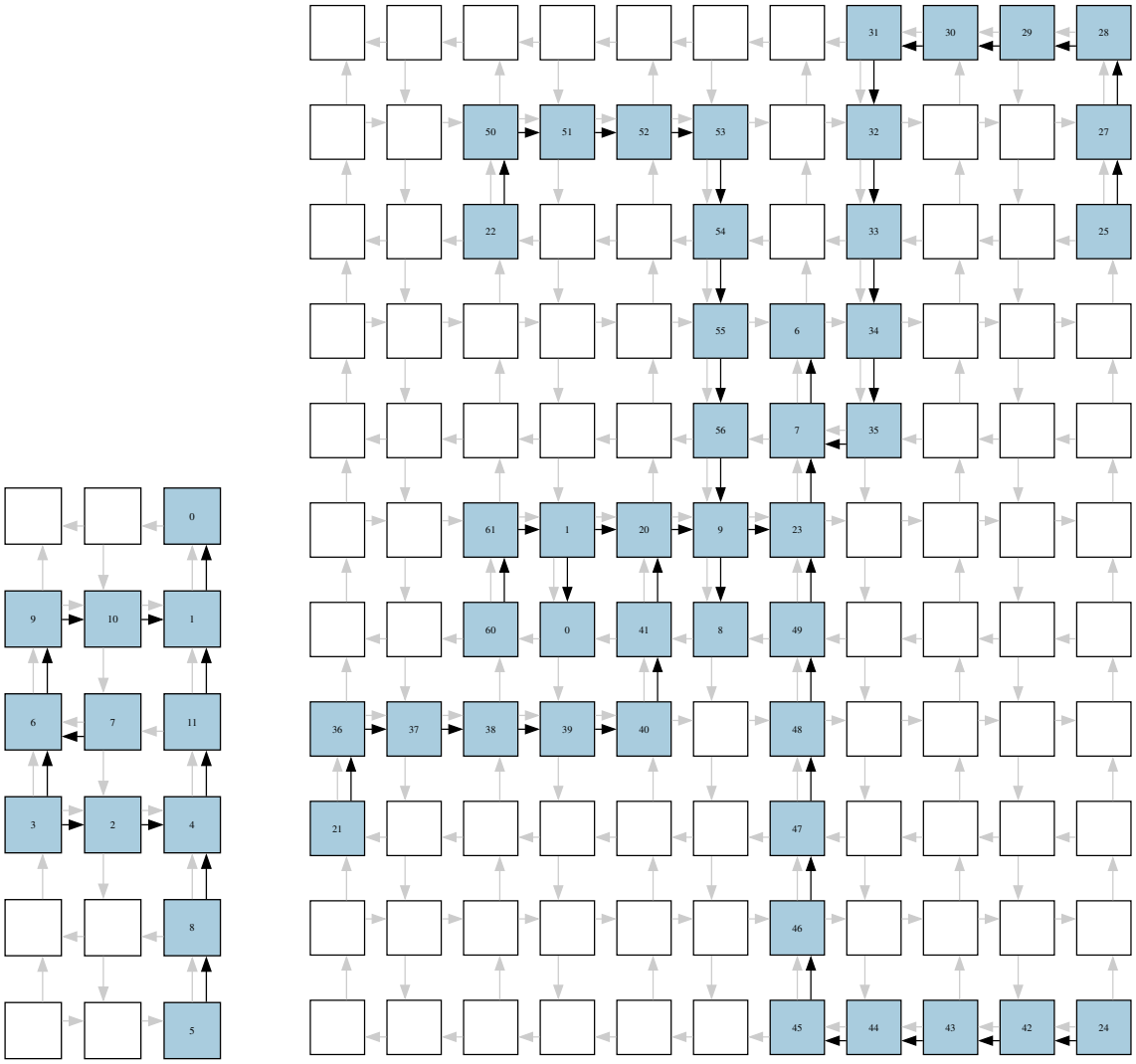
espaço.

Um exemplo ilustrativo desse comportamento é apresentado na Figura 4.19, na qual se observa o posicionamento final obtido para o *benchmark 2:1 MUX*. Nessa configuração, os nós 0 e 13 encontram-se alocados em posições afastadas da região central do circuito. No entanto, ambos poderiam ser posicionados em células imediatamente acima daquela que abriga o nó 1. Esse rearranjo espacial permitiria a redução da área total ocupada de 25 para 15 células. Esse exemplo mostra a necessidade de aperfeiçoamento da função de custo empregada no algoritmo SA, de modo a incorporar métricas diretamente associadas à compactação do *layout*.

À medida que o desenvolvimento do algoritmo YOTT avança, foram selecionados dois exemplos ilustrativos de circuitos posicionados por essa abordagem. Os exemplos correspondem aos *benchmarks 2:1 Mux* e *CLPL*, sendo o primeiro implementado com duas camadas adicionais de interconexões e o segundo sem a utilização de camadas extras de fios.

Na Figura 4.20, observa-se que, no caso do *2:1 Mux*, ilustrado em (a), o algoritmo YOTT alcança um posicionamento mais eficiente em termos de área quando comparado ao algoritmo SA, mesmo considerando um número semelhante de camadas adicionais. Por sua vez, no caso do *benchmark CLPL*, apresentado em (b), o YOTT produz um posicionamento de custo zero, enquanto o algoritmo SA não conseguiu uma solução com custo equivalente para o mesmo circuito e sob a mesma configuração. Ressalta-se, adicionalmente, que o YOTT adota um mecanismo de operação mais simples e não possui como objetivo primário a compactação do circuito.

Adicionalmente, a inspeção visual do espaço ocupado evidencia oportunidades de compactação que não são exploradas pelo algoritmo. Por exemplo, os nós 21, 36 e 37



(a) *Benchmark 2:1 Mux* com duas camadas extras.

(b) *Benchmark CLPL*.

Figura 4.20: Posicionamentos obtidos pelo YOTT para o *benchmark 2:1 Mux* com 2 camadas de fios extras (a) e *CLPL* com nenhuma camada extra (b).

podem ser realocados para as células imediatamente abaixo do nó 38, o que possibilita uma otimização vertical da matriz. De forma análoga, os nós 29, 28, 27 e 25 apresentam potencial de reposicionamento mais eficiente em células imediatamente abaixo do nó 30. Por fim, os nós 45, 44, 43, 42 e 24 podem ser deslocados para células à esquerda do nó 46. Essas modificações permitiriam a redução da área total requerida para o posicionamento do *benchmark CLPL* de 121 para 70 células, correspondentes a uma matriz de 7 colunas por 10 linhas.

Capítulo 5

Trabalhos Relacionados

Este capítulo apresenta os trabalhos da literatura relacionados ao problema de posicionamento de grafos em matrizes. Inicialmente, na Seção 5.1, discutem-se trabalhos que investigaram a implementação em *hardware* de algoritmos de posicionamento. Em seguida, a Seção 5.2 aborda estudos relacionados ao posicionamento em arquiteturas CGRAs. A Seção 5.3 apresenta os trabalhos desenvolvidos para o posicionamento em FPGAs, enquanto a Seção 5.4 trata das abordagens propostas para o posicionamento em QCAs.

5.1 Posicionamento em *hardware*

Os primeiros posicionadores desenvolvidos diretamente em *hardware* foram propostos em 1983 por Chyan e Breuer [21]. Os autores propuseram uma nova abordagem para acelerar o problema de posicionamento por meio de *hardware* dedicado, capaz de explorar o paralelismo espacial. O sistema baseou-se em um arranjo sistólico de elementos de processamento (PEs) em uma grade, na qual cada PE era responsável por uma posição física do chip e executava trocas locais com seus vizinhos. As decisões de troca eram guiadas por uma função de custo, enquanto um barramento global era empregado para a propagação serial das novas localizações das células após as trocas. Relataram que esse arranjo permitiu acelerar o processo de posicionamento em $O(n)$, onde n representa o número de PEs.

Essa base conceitual foi estendida ao domínio dos FPGAs por Wrighton e DeHon em 2003 [97]. Os autores propuseram um acelerador sistólico capaz de executar o posicionamento em milissegundos ao adaptar o algoritmo de *Simulated Annealing*. A principal inovação consistiu em permitir que as informações de posição permanecessem temporariamente desatualizadas entre atualizações globais periódicas, o que viabilizou uma aceleração de duas a três ordens de magnitude em relação à abordagem serial anterior

Posteriormente, surgiram propostas que exploraram a integração entre arquiteturas de propósito geral e aceleradores especializados. Dhar et al. [26] apresentaram uma abordagem híbrida que combina CPU e FPGA para paralelizar o cálculo do gradiente

descendente no posicionamento analítico, com ganhos de aproximadamente 33× em relação a implementações sequenciais em CPU. Contudo, essa técnica restringe-se à etapa de posicionamento global, não contemplando a fase de legalização das posições das LUTs no dispositivo.

Posteriormente, Vieira et al. [82] propuseram o *RESHAPE*, uma arquitetura de mapeamento em tempo de execução para CGRAs baseada em FPGAs. A proposta utiliza como entrada um grafo DFG, implementada por meio de máquinas de estados e travessias sequenciais do grafo, com o objetivo de reduzir o tempo de mapeamento durante a execução da aplicação. Embora apresente ganhos significativos de desempenho em relação a soluções puramente em software, o *RESHAPE* limita a exploração do paralelismo temporal devido à natureza sequencial de sua lógica de controle.

Em contraste, o acelerador desenvolvido nesta tese [67] apresenta uma implementação completa do algoritmo de *Simulated Annealing* em *hardware* e explora o paralelismo temporal por meio de uma arquitetura em *pipeline* associada à execução de múltiplas *threads*. Diferentemente das propostas sistólicas de Wrighton e DeHon [97, 74], nas quais a execução de cada tentativa de troca requer a atualização de todos os PEs, com custo proporcional ao tamanho do *array*, o acelerador proposto executa uma iteração completa do SA em dois ciclos de *clock*. Além disso, a atualização do estado global é desacoplada das tentativas locais de troca, evitando a necessidade de sincronização completa do *array* a cada nova iteração.

Em paralelo, outros trabalhos estabeleceram as bases matemáticas do posicionamento por meio da definição de funções de custo e formulações analíticas do problema. Kleinhans et al. [45] introduziram o método GORDIAN, que formulou o posicionamento global como um problema contínuo de otimização, estabelecendo fundamentos para abordagens analíticas posteriores. No contexto de FPGAs, Xu e Khalid [100] exploraram explicitamente o uso de funções de custo quadráticas, demonstrando sua eficiência computacional e adequação à natureza regular.

Desenvolvimentos posteriores estenderam essas formulações para arquiteturas heterogêneas. Gort e Anderson [36] propuseram o HeAP, um posicionador analítico para FPGAs contendo múltiplos tipos de recursos. Mais recentemente, Meng et al. [52] apresentaram o elfPlace, que emprega um modelo eletrostático para escalonar o posicionamento analítico em dispositivos de grande porte. Rajarathnam et al. [68], por sua vez, introduziram o DREAMPlaceFPGA, que explora GPUs e bibliotecas de aprendizado profundo para acelerar o processo analítico. Apesar dos avanços em escalabilidade e qualidade, tais abordagens concentram-se predominantemente no posicionamento global e mantêm dependência de etapas adicionais de legalização e refinamento.

Em paralelo às evoluções nos algoritmos de posicionamento, outros trabalhos investigaram a aceleração do fluxo de compilação por meio da decomposição do design

em partes menores. Guo et al. [37] propuseram o RapidStream, que divide aplicações descritas em HLS em múltiplas páginas fisicamente independentes, permitindo a implementação paralela de cada partição. De forma semelhante, Xiao et al. [98] introduziram o fluxo PLD, que visa compatibilizar a compilação rápida de FPGAs com processos modernos de refinamento incremental.

Abordagens anteriores de reconfiguração parcial automatizada, como a proposta por Vipin e Fahmy [83], já indicavam o potencial da decomposição espacial do *fabric* para reduzir o tempo de compilação. Entretanto, mesmo nesses fluxos, cada partição individual permanece sujeita aos custos elevados das etapas tradicionais de posicionamento, limitando a obtenção de latências compatíveis com aplicações dinâmicas ou em tempo de execução.

A avaliação comparativa de técnicas de posicionamento e compilação acelerada baseia-se, em geral, em ferramentas consolidadas e conjuntos de *benchmarks* representativos. O VTR [29] constitui a principal plataforma de referência para a exploração arquitetural e a avaliação de algoritmos de posicionamento em FPGAs, sendo amplamente utilizada como *baseline* experimental.

5.2 Posicionamento em CGRAs

Esta seção discute técnicas propostas para o posicionamento e mapeamento de CGRAs, com ênfase no tempo de execução e na escalabilidade. De forma geral, as abordagens mais rápidas baseiam-se em heurísticas, como técnicas de travessia de grafos e *Simulated Annealing* (SA), enquanto métodos exatos apresentam limitações práticas decorrentes da complexidade do problema. Em particular, soluções exatas tendem a restringir-se a grafos com um pequeno número de vértices, geralmente menores que 30, e a tempos de execução que variam de minutos a horas. Mais recentemente, surgiram abordagens baseadas em aprendizado profundo que, apesar de inicialmente apresentarem custos computacionais elevados, passaram a alcançar tempos de execução da ordem de segundos em propostas mais recentes.

Os métodos exatos formulam o problema de mapeamento como instâncias de otimização combinatória, normalmente por meio da programação linear inteira. Um exemplo é o CGRA-ME [85], que emprega essa formulação, mas apresenta limitações de escalabilidade e pode demandar minutos para a obtenção de uma solução. Extensões posteriores adicionaram estruturas baseadas em ZBDD [10], o que reduziu o tempo de execução para a ordem de centenas de milissegundos. Outras propostas utilizam solucionadores SAT [76], embora permaneçam restritas pela complexidade intrínseca do problema e pela escalabilidade limitada.

Nos últimos anos, observa-se o crescimento de abordagens baseadas em aprendizado profundo para o mapeamento em CGRAs [54, 46, 102, 53, 47, 61, 51]. Liu

et al. [54] apresentaram um agente de Aprendizado por Reforço (RL) cujo tempo médio de mapeamento situa-se na ordem de minutos. Kojima et al. [46] propuseram a combinação de redes neurais convolucionais sobre grafos (GCN) com algoritmos genéticos, enquanto Zhuang et al. [102, 53] exploraram o uso de redes neurais de grafos (GNN). Embora essas abordagens representem avanços relevantes, seus tempos de execução variam de segundos a horas.

Propostas mais recentes buscam reduzir esse custo computacional. Li et al. [51] introduziram uma técnica baseada em GNN para gerar uma incorporação geométrica restrita do grafo, alcançando tempos de execução de poucos segundos. De forma complementar, Kong et al. [47] combinaram redes de atenção em grafos com RL. Adicionalmente, Mohtavipour et al. [61] apresentaram uma abordagem híbrida que emprega GCN para particionar o grafo em sub-regiões da malha e aplica SA para o posicionamento de cada subgrafo.

Embora abordagens baseadas em SA estejam entre as primeiras propostas para o mapeamento em CGRAs [59], elas permanecem relevantes tanto nesse domínio quanto no contexto de FPGAs. Nesse cenário, o VPR [29] consolida-se como o estado da arte acadêmico em posicionamento e roteamento, adotando o SA como técnica principal na etapa de posicionamento.

Entre as abordagens heurísticas baseadas em grafos, destacam-se aquelas que apresentam os menores tempos de execução, variando de milissegundos a segundos [48, 101, 20, 15, 95]. Lai et al. [48] propuseram uma estratégia de travessia baseada em árvore geradora mínima, utilizando o algoritmo de Kruskal, com complexidade $O(V^2)$. Entretanto, essa técnica tende a dispersar os vértices pela malha e resulta em uma baixa compactação. Yoon et al. [101] introduziram o método *split and push*, que inicia com todos os nós posicionados em uma mesma coordenada e realiza divisões sucessivas do grafo seguidas de realocação espacial até que cada vértice ocupe uma posição distinta, alcançando tempos de mapeamento da ordem de segundos para grafos com até 30 nós. Essa abordagem foi posteriormente estendida por Wijerathne et al. [95], que incorporaram técnicas de clusterização espectral para particionar grafos maiores em subgrafos, aos quais se aplica uma versão modificada do algoritmo *split and push*.

Uma das propostas mais próximas da presente tese é o trabalho RESHAPE, apresentado por Vieira et al. [82]. Os autores propõem uma abordagem de posicionamento e roteamento para CGRAs implementada diretamente em FPGA, na forma de um *overlay* com múltiplas unidades de mapeamento replicadas. A estratégia baseia-se em um algoritmo guloso derivado de travessia de grafos, implementado como máquinas de estados finitos, permitindo a execução paralela de múltiplas tentativas de posicionamento. A proposta adota um modelo incremental, no qual novas soluções podem substituir a configuração corrente da CGRA durante a operação. Os resultados experimentais demonstram reduções de ordens de magnitude no tempo de posicionamento

quando comparado a ferramentas em software, como o VPR e o CGRA-ME.

Diferentemente do RESHAPE [82], que propõe um mecanismo determinístico de mapeamento incremental baseado em máquina de estados para arquiteturas CGRA, o acelerador desenvolvido nesta tese implementa o algoritmo completo de *Simulated Annealing* diretamente em *hardware*. A proposta explora o paralelismo temporal por meio de sua arquitetura projetada em pipeline e da execução simultânea de múltiplas instâncias do algoritmo, o que permite a exploração concorrente do espaço de soluções. Enquanto o RESHAPE realiza uma única trajetória de mapeamento dependente da ordem de visita do grafo, o acelerador proposto possibilita o refinamento global da solução, sem comprometer a latência de execução. Além disso, a arquitetura apresentada é genérica e aplicada a diferentes domínios de posicionamento, como CGRAs, FPGAs e QCA.

A Tabela 5.1 sintetiza as principais características dos trabalhos discutidos nesta seção. A comparação mostra que os métodos exatos fornecem referências de qualidade, mas apresentam limitações de escalabilidade. As abordagens baseadas em aprendizado profundo ampliam o espaço de exploração e podem reduzir o tempo de mapeamento em propostas mais recentes, embora dependam de modelos de aprendizado e não tenham como foco principal a implementação direta de unidades de posicionamento em *hardware*. Por sua vez, as heurísticas baseadas em grafos apresentam menor custo computacional, mas tendem a depender fortemente da estratégia de travessia e da estrutura do grafo.

O trabalho RESHAPE [82] representa a proposta mais próxima desta tese no domínio dos CGRAs, por também investigar o mapeamento em tempo de execução com suporte em FPGA. Entretanto, a abordagem emprega uma lógica de controle baseada em máquinas de estados e uma estratégia gulosa de mapeamento incremental. Em contraste, a presente tese implementa o algoritmo completo de *Simulated Annealing* em *hardware* e também adapta os algoritmos YOTO e YOTT para uma organização em *pipeline*, com execução simultânea de múltiplas instâncias.

5.3 Posicionamento em FPGAs

Esta seção revisa trabalhos que contextualizam a pesquisa atual sobre o posicionamento de FPGAs, com abrangência nas abordagens desenvolvidas em *software*, propostas de aceleração por *hardware* e nos métodos recentes que utilizam aprendizado de máquina (ML).

Wrighton e DeHon[97] (já citados na Seção 5.2) propuseram um acelerador baseado em *Simulated Annealing* (SA), implementado como um arranjo sistólico, no qual múltiplas trocas locais são avaliadas em paralelo. Embora esse trabalho represente um marco conceitual ao demonstrar a viabilidade do posicionamento acelerado em *hardware*, sua

Tabela 5.1: Comparação entre trabalhos relacionados sobre mapeamento e posicionamento em CGRAs.

Trabalho	Abordagem	Característica principal	Limitação ou diferença	Relação com esta tese
Walker e Anderson [85], Beidas e Anderson [10], Tirelli et al. [76]	Métodos exatos baseados em ILP, ZBDD e SAT	Formulação do mapeamento como problema de otimização combinatória.	Fornecem soluções ou referências de qualidade, mas apresentam limitações de escalabilidade para grafos maiores.	Esta tese adota heurísticas e meta-heurísticas compatíveis com execução em *hardware*, com foco na redução do tempo de execução.
Liu et al. [54], Kojima et al. [46], Zhuang et al. [102], Li et al. [53], Kong et al. [47], Mohtavipour et al. [61], Li et al. [51]	Aprendizado profundo, RL, GCN e GNN	Uso de modelos de aprendizado para orientar o mapeamento, particionar grafos ou gerar representações geométricas.	Apresentam avanços relevantes, mas não têm como foco principal uma unidade dedicada de posicionamento em *hardware* baseada em *pipeline*.	Esta tese explora algoritmos simples e parametrizáveis, com implementação direta em *hardware* e execução concorrente de múltiplas instâncias.
Mei et al. [59]	Abordagem baseada em SA	Uso de SA como técnica de posicionamento ou mapeamento, com boa capacidade de exploração do espaço de soluções.	O SA apresenta custo computacional elevado.	Esta tese implementa o SA em *hardware*, com múltiplas *threads* em *pipeline*, de modo a reduzir a latência por iteração.
Lai et al. [48], Yoon et al. [101], Chen e Mitra [20], Canesche et al. [15], Wijerathne et al. [95]	Heurísticas baseadas em grafos	Estratégias de travessia, particionamento e decomposição para reduzir o tempo de mapeamento.	Apresentam baixo custo computacional, mas a qualidade das soluções depende da ordem de travessia, da estrutura do grafo e das decisões locais.	Esta tese segue um caminho semelhante, porém focado execução em *hardware* execução com múltiplas instâncias paralelas.
Vieira et al. [82]	Mapeamento em tempo de execução implementado em FPGA	Proposta de um *overlay* para CGRAs, com múltiplas unidades de mapeamento e estratégia gulosa baseada em travessia de grafos.	A abordagem é baseada em máquinas de estados e em mapeamento incremental determinístico, sem implementação completa do SA.	É uma das propostas mais próximas desta tese; contudo, esta tese explora SA, YOTO e YOTT em uma arquitetura genérica baseada em *pipeline* e múltiplas *threads*.
Esta tese	Aceleradores em *hardware* para SA, YOTO e YOTT	Organização em *pipeline*, execução de múltiplas *threads*, uso de memórias distribuídas e adaptação a diferentes domínios de posicionamento.	Não busca substituir métodos exatos como referência de otimização, nem técnicas de aprendizado profundo como estratégia de exploração global.	Propõe uma arquitetura genérica de aceleração para algoritmos iterativos e baseados em travessia, aplicada a CGRAs, FPGAs e QCA.

arquitetura apresenta limitações de escalabilidade para *designs* modernos, além de um tempo de execução elevado, decorrente da serialização da etapa de atualização global.

A avaliação de novas técnicas de posicionamento apoia-se, em geral, em ferramentas acadêmicas consolidadas. O VPR [57] e sua evolução mais recente, o VTR 9 [29], representam o estado da arte em ferramentas de CAD de código aberto para FPGAs e são utilizados como referência para a comparação da qualidade de posicionamento e do tempo de execução.

Entre os algoritmos heurísticos, Ferreira et al. [32] propuseram um método polinomial baseado em grafos para posicionamento e roteamento em FPGAs, que serve

como base para estratégias gulosas adotadas e aprimoradas em trabalhos posteriores. Abordagens analíticas e paralelizadas também foram exploradas, como no trabalho de Dhar et al. [26], que propõe um posicionamento analítico *multithread*. Apesar dos ganhos obtidos, essa solução mostrou-se mais lenta do que abordagens baseadas em hardware dedicado e limita-se à etapa de posicionamento global.

Trabalhos recentes investigaram o uso de técnicas de aprendizado de máquina para aprimorar o posicionamento em FPGAs. Abordagens baseadas em Aprendizado por Reforço (RL), como as propostas por Wang et al. [91] e Tian et al. [75], demonstram melhorias na qualidade das soluções obtidas, embora exijam longos períodos de treinamento em GPUs. Métodos híbridos também foram explorados, como a proposta de Rajarathnam et al. [69], que combina formulações analíticas com SA para refinar o posicionamento. Apesar dos avanços, essas abordagens apresentam custos computacionais elevados e dependem de infraestrutura de treinamento.

A Tabela 5.2 organiza os trabalhos desta seção na qual são mostradas as principais contribuições, limitações e comparações em relação aos trabalhos desta tese. Pela comparação, observa-se que os trabalhos relacionados avançam em algumas direções como métodos exatos ou heurísticos voltados ao posicionamento e roteamento, propostas de aceleração por *hardware* e abordagens recentes baseadas em aprendizado de máquina. Nesta tese são investigados algoritmos de posicionamento com estrutura mais simples e maior potencial de execução acelerada com a manutenção do compromisso entre qualidade da solução, caminho crítico e tempo de execução.

5.4 Posicionamento em QCAs

Nesta seção, são discutidos os trabalhos relacionados ao posicionamento de grafos de circuitos na tecnologia QCA. A revisão apresentada em Wang et al. [92] indica que, desde a introdução do conceito de QCA em 1993, inicialmente motivada pela superação das limitações físicas da tecnologia CMOS, o interesse acadêmico permaneceu relativamente restrito até aproximadamente 2015, quando surgiram os primeiros algoritmos heurísticos voltados ao posicionamento de circuitos QCA. A partir desse período, as pesquisas passaram a concentrar-se no desenvolvimento de métodos heurísticos para síntese, posicionamento e roteamento, com destaque para abordagens baseadas em Algoritmos Genéticos, Redes Neurais Artificiais e *Support Vector Machines*, cujo objetivo principal consiste na redução da área total e do número de portas. Embora métodos exatos tenham permitido a obtenção de soluções ótimas em termos de área mínima para fins de comparação, a baixa escalabilidade dessas abordagens limita sua aplicação prática. Como tendências, o estudo aponta a necessidade de maior robustez e confiabilidade para viabilizar a fabricação de circuitos QCA, bem como a investigação de estratégias que flexibilizem o sincronismo global, responsável por comprometer até

Tabela 5.2: Relação dos trabalhos relacionados ao problema de P&R em FPGAs.

Trabalho	**Principal contribuição**	**Limitação ou aspecto não explorado**	**Relação com esta tese**
Wrighton e DeHon [97]	Propõe um acelerador em *hardware* baseado em SA, organizado como um arranjo sistólico.	Apresenta limitações de escalabilidade para *designs* modernos e custo associado à atualização global serializada.	Esta tese também investiga aceleração do posicionamento, mas explora uma organização em *pipeline*, múltiplas *threads* e algoritmos baseados em travessia de grafos.
VPR [57] e VTR 9 [29]	Fornecem ferramentas consolidadas de CAD para exploração de arquiteturas e posicionamento em FPGAs.	São usados como referência de qualidade e tempo de execução, mas não têm como objetivo principal o posicionamento acelerado em *hardware*.	Foram utilizadas como base de comparação para a avaliação do compromisso entre qualidade da solução e tempo de execução.
Ferreira et al. [32]	Propõe um algoritmo polinomial baseado em grafos para posicionamento e roteamento em FPGAs virtuais.	O trabalho base para estratégias gulosas, mas não explora a organização arquitetural proposta nesta tese.	Propõe a implementação em *hardware* dos algoritmos gulosos baseados em travessia de grafos.
Dhar et al. [26]	Propõe uma abordagem analítica e paralelizada para posicionamento em FPGAs.	Limita-se à etapa de posicionamento global e, conforme discutido, apresenta desempenho inferior ao de abordagens com *hardware* dedicado.	Nesta tese são investigadas alternativas voltadas à redução do tempo de execução por meio de uma organização mais diretamente associada à execução em *hardware*.
Wang et al. [91] e Tian et al. [75]	Aplicam Aprendizado por Reforço ao problema de posicionamento em FPGAs.	Melhoram a qualidade das soluções, mas exigem longos períodos de treinamento em GPUs.	Nesta tese não há dependência de treinamento prévio e busca reduzir o tempo de execução por meio da estrutura dos algoritmos e de sua organização arquitetural.
Rajarathnam et al. [69]	Combina métodos analíticos com SA para refinar o posicionamento.	Apresenta custos computacionais elevados e dependência de infraestrutura de treinamento ou execução mais complexa.	Esta tese também considera SA, mas o avalia no contexto de uma proposta voltada à aceleração e ao compromisso entre qualidade e tempo de execução.
Esta tese	Adapta os algoritmos SA, YOTO e YOTT ao posicionamento de LUTs, com a utilização de estruturas de memória e organização em *pipeline*.	Os resultados sugerem uma relação entre tempo de execução e qualidade da solução, com possível degradação do caminho crítico em alguns casos.	Diferencia-se por investigar algoritmos de posicionamento com foco em execução acelerada e em aplicações que demandam rápida adaptação do *hardware* com bons ganhos em relação ao tempo de execução.

70% da área ocupada.

Walter et al. [88] propuseram um algoritmo exato para o posicionamento de circuitos QCA sob o esquema de *clock* USE, baseado em um solucionador SMT. O objetivo do trabalho consistiu no estabelecimento de uma referência de qualidade para comparação, e não na obtenção de tempos de execução reduzidos. Os autores demonstraram que abordagens heurísticas anteriores produziam soluções significativamente afastadas do ótimo e com até três vezes a área ocupada. Embora o tempo de execução do método exato cresça rapidamente com o aumento do tamanho do circuito, para instâncias de pequeno porte, ele apresenta desempenho semelhante ao de heurísticas existentes.

Em Fontes et al. [35], foi proposto um algoritmo heurístico para o posicionamento e roteamento de circuitos QCA com foco na redução da área ocupada. A abordagem baseia-se na decomposição do grafo de conexões em subgrafos associados a regiões reconvergentes, que são posicionados de forma independente em matrizes distintas. Em uma etapa posterior, um segundo algoritmo recompõe o grafo original a partir dos subgrafos posicionados. Os resultados experimentais indicaram reduções de até 50% na área ocupada em relação a métodos anteriores, embora o trabalho não apresente uma análise detalhada do impacto sobre o tempo de execução. Os algoritmos apresentados em [88, 35] serviram como base para o estudo conduzido em [78].

Torres et al. [78] discutiram os principais desafios associados à utilização de QCA em circuitos digitais, com ênfase na necessidade de sincronização dos sinais por meio de esquemas de *clock* e no compartilhamento do mesmo plano físico por portas lógicas e fios. Essa característica conduz à utilização de até 70% da área ocupada no balanceamento do grafo. Os autores avaliaram a implementação de circuitos QCA sob o esquema de *clock* USE, utilizando o algoritmo heurístico proposto em [35], e compararam os resultados com o método exato apresentado em [88]. Os experimentos mostraram que a remoção do sincronismo possibilita reduções de área de até 67%, com médias de 33% para o método heurístico e 19% para o método exato. Contudo, essa redução ocorre à custa da diminuição da vazão do circuito, uma vez que a execução correta exige a sincronização dos sinais em etapas posteriores.

Além dos trabalhos que abordam o fluxo completo de posicionamento e roteamento, Walter et al. [87] propuseram um algoritmo de roteamento projetado para operar de forma independente do posicionador, com o objetivo de aumentar a escalabilidade do fluxo de projeto. Os resultados experimentais indicaram um desempenho de ordens de magnitude superior ao de ferramentas do estado da arte, o que permitiu a geração de roteamentos para *layouts* complexos em menos de um segundo, além de fornecer diagnósticos precisos ao identificar conexões insatisfatórias.

Como aprimoramento, Walter et al. [86] apresentaram uma extensão do algoritmo *Ortho*, reconhecido como um dos métodos meta-heurísticos mais rápidos para posicionamento e roteamento em QCA. As melhorias concentraram-se na reorganização

da disposição dos fios e das conexões de entrada e saída das portas lógicas, com o objetivo de reduzir o número de cruzamentos por meio de técnicas de ordenação de sinais. Adicionalmente, foi introduzido o suporte a portas *maioria* de três entradas. Os resultados indicaram reduções de aproximadamente 25% na área ocupada e de cerca de 17% no número de cruzamentos em relação à implementação original do *Ortho*. A adoção das portas *maioria* possibilitou reduções de até 46% nos cruzamentos; entretanto, o aumento da complexidade do roteamento implicou um crescimento da área total da ordem de 1000%. Diferentemente dos trabalhos citados, as propostas desta tese são concebidas desde a origem para execução em aceleradores de *hardware*, enquanto as abordagens da literatura se concentram em implementações puramente em software.

Mais recentemente, Li et al. [50] propuseram um algoritmo híbrido para o posicionamento e roteamento em QCA, com o objetivo de superar limitações observadas em abordagens anteriores, tais como baixas taxas de sucesso e elevados tempos de execução. A proposta emprega o algoritmo de *Simulated Annealing* (SA), respeitando as restrições de sincronização dos esquemas de *clock*. Os resultados experimentais mostram taxas de sucesso superiores a 90% e reduções no tempo de execução da ordem de 400% a 500% em esquemas de *clock* bidimensionais, como USE e *2D-Wave*. Na abordagem proposta por Li et al., cada iteração do SA realiza perturbações aleatórias nas coordenadas dos nós do grafo, seguidas do cálculo da função de aptidão (*fitness*). A decisão de aceitação da nova solução segue o critério probabilístico do SA. A função de custo combina fatores como o número de cruzamentos, o respeito às restrições de *clock* e a área ocupada. Para a avaliação da viabilidade das interconexões, a abordagem recorre adicionalmente à execução do algoritmo A^*, utilizado para a verificação explícita dos caminhos de roteamento.

A Tabela 5.3 organiza os trabalhos discutidos nesta seção a em comparação com o proposto. Além de indicar a principal contribuição de cada abordagem, a tabela mostra as limitações ou aspectos não explorados em cada trabalho.

A comparação mostra que os trabalhos relacionados se direcionam principalmente na obtenção de soluções ótimas, na redução de área, na análise do impacto do sincronismo e no aprimoramento do roteamento. A contribuição desta tese situa-se em outro ponto: a simplificação da decisão local do SA, por meio de uma função de custo baseada em penalidades e bonificações, e a organização dos algoritmos de posicionamento de modo compatível com sua futura implementação em *hardware*.

Tabela 5.3: Relação dos trabalhos relacionados ao problema de P&R em QCA.

Trabalho	**Principal contribuição**	**Limitação ou aspecto não explorado**	**Relação com esta tese**
Wang et al. [92]	Sistematiza métodos meta-heurísticos aplicados à síntese, posicionamento e roteamento em QCA.	Não propõe um novo posicionador; identifica limitações de escalabilidade, sincronismo global, robustez e confiabilidade.	Contextualiza a área e reforça a necessidade de soluções mais escaláveis.
Walter et al. [88]	Propõe um método exato baseado em SMT para posicionamento em QCA sob o esquema USE.	Fornece referência de qualidade, mas o tempo de execução cresce rapidamente com o tamanho dos circuitos.	Não se busca uma solução exata, mas uma abordagem com menor complexidade por iteração e potencial de aceleração em *hardware*.
Fontes et al. [35]	Propõe uma heurística baseada na decomposição do grafo em subgrafos reconvergentes para reduzir a área ocupada.	Não apresenta análise detalhada do impacto da abordagem sobre o tempo de execução.	Nesta tese também é considerada a área, mas é enfatizada uma função de custo compatível com a utilização futura em aceleradores em *hardware*.
Torres et al. [78]	Avalia o impacto do sincronismo e das interconexões em circuitos QCA sob o esquema USE.	Mostra que a remoção do sincronismo reduz área, mas compromete a vazão do circuito.	Serve como base comparativa para os experimentos desta tese e evidencia a importância de considerar restrições de sincronização.
Walter et al. [87]	Propõe um algoritmo de roteamento multipercurso independente do posicionador.	Trata especificamente o roteamento, sem atuar como um posicionador integrado.	Nesta tese, o posicionamento em QCA é considerado de forma integrada ao roteamento, a partir de grafos balanceados.
Walter et al. [86]	Apresenta redes versáteis de distribuição de sinais, com redução de área e cruzamentos em relação ao *Ortho*.	O aumento da complexidade do roteamento pode elevar significativamente a área total.	Nesta tese o foco foi na adaptação de algoritmos de posicionamento para uma organização compatível com aceleração em *hardware*.
Li et al. [50]	Propõe um algoritmo híbrido baseado em SA para posicionamento e roteamento em QCA.	Utiliza o algoritmo A^* para verificar explicitamente os caminhos de roteamento.	Esta tese também utiliza SA, mas propõe uma função de custo baseada em penalidades e bonificações que dispensa a execução de algoritmos adicionais de busca de caminhos.
Esta tese	Propõe uma função de custo específica para o SA e discute a adaptação de YOTO e YOTT ao domínio QCA.	Os resultados em QCA ainda são preliminares e não consideram cruzamento de fios nem derivação de sinais em uma mesma célula.	Diferencia-se por tratar posicionamento e roteamento de forma integrada e por orientar a proposta à futura implementação em aceleradores de *hardware*.

Capítulo 6

Conclusão e Trabalhos Futuros

6.1 Considerações Finais

A questão fundamental que orientou esta tese consiste em investigar se o processo de posicionamento de grafos pode ser acelerado por meio da implementação dedicada de algoritmos em *hardware* reconfigurável.

A resposta a essa questão foi construída progressivamente ao longo dos capítulos desta tese. Inicialmente, foi estabelecida a base teórica do problema de posicionamento bidimensional, com destaque para a sua complexidade computacional e suas particularidades em diferentes domínios. Foram selecionados três algoritmos da literatura: Simulated Annealing (SA), YOTO e YOTT; com o objetivo de investigar tanto uma meta-heurística consolidada quanto abordagens baseadas em travessia de grafos, estruturalmente mais simples e potencialmente mais adequadas à implementação em *hardware*.

Com isso, desenvolveu-se a principal contribuição da tese: um modelo genérico de aceleradores para algoritmos iterativos e estocásticos de posicionamento. A proposta fundamentou-se na decomposição do algoritmo em estágios *pipeline*, com múltiplas *threads* independentes e uso de memórias distribuídas baseadas em B-RAM. Essa organização é capaz de explorar o paralelismo temporal e espacial de forma combinada, com o objetivo de reduzir a latência por iteração e aumentar a taxa de exploração do espaço de soluções. Introduziu-se, ainda, o conceito de *pipeline* vertical e a utilização de *caches* distribuídas, com foco na redução da utilização de recursos e na ampliação da escalabilidade da arquitetura. O modelo foi concebido de modo a permitir sua adaptação a diferentes domínios com restrições distintas.

Após as propostas, procedeu-se à validação experimental da proposta em três contextos arquiteturais: CGRAs, FPGAs e QCA. Para o domínio dos CGRAs, foram implementadas unidades aceleradoras em RTL para os algoritmos SA, YOTO e YOTT. Os resultados indicaram que a organização em *pipeline* com múltiplas *threads* foi capaz de reduzir a latência do processo de posicionamento com a preservação da qualidade da solução compatível com as versões de referência em *software*. A análise do consumo de recursos, da utilização de memória e da vazão de dados mostrou

também a viabilidade prática da abordagem em dispositivos FPGA reais.

No domínio dos FPGAs, os algoritmos foram adaptados ao posicionamento de LUTs. A reestruturação dos *pipelines*, a reorganização das estruturas de memória e a introdução da *cache* permitiram reduzir o consumo de B-RAM e viabilizar a implementação paralela de grafos maiores. Os resultados experimentais mostram que os algoritmos YOTO e SA apresentam potencial de melhoria de desempenho e evidenciam um *trade-off* entre a qualidade das soluções e o tempo de execução. Enquanto o algoritmo SA produz soluções de qualidade ao custo de um maior esforço computacional, os algoritmos baseados em travessia geram soluções próximas do ótimo com um tempo de execução reduzido, mostrando-se mais adequados para aplicações que demandam respostas rápidas ou mapeamento *on-the-fly*. Ainda que, em determinados casos, se observe uma degradação no caminho crítico da solução em comparação com a ferramenta de referência, essa troca entre tempo de execução e qualidade mostra um espaço de projeto para aplicações que necessitem de uma rápida adaptação do *hardware*.

Para a tecnologia QCA, a arquitetura proposta foi estendida para lidar com restrições de interconexão e de sincronização mais rígidas.Foi introduzida uma função de custo específica para o SA, com penalidades associadas à adjacência, à orientação das conexões e às restrições topológicas impostas pelo esquema de sincronização.. Os resultados preliminares mostraram que o modelo de acelerador possui potencial para futura aplicação.

6.2 Trabalhos Futuros

Os resultados obtidos para o posicionamento em arquiteturas FPGA indicam que os algoritmos propostos apresentam potencial. Apesar dos avanços alcançados, permanecem desafios relevantes a serem abordados como continuidade deste trabalho. Um primeiro eixo de investigação consiste no aprimoramento do algoritmo YOTT. A estratégia atual de anotação e convergência topológica não produziu ganhos consistentes, sobretudo em circuitos de maior complexidade. Como trabalho futuro, pretende-se revisar os critérios de anotação empregados.

Outro eixo refere-se à implementação parcial dos algoritmos de posicionamento em FPGA, pois foram mapeados somente as versões para CGRA. Essa etapa permitirá uma análise detalhada do uso de recursos, em especial das estruturas de mapeamento N_2C e C_2N, bem como do impacto do uso de memórias com suporte a *cache* distribuído.

No que se refere à organização espacial do posicionamento de FPGA, os resultados indicaram que, à medida que a taxa de ocupação da matriz de LUTs se aproxima de valores elevados, o problema deixa de apresentar caráter predominantemente local e passa a manifestar efeitos globais de congestionamento.

Diante desse comportamento, propõe-se como trabalho futuro a reformulação do problema de posicionamento por meio do particionamento da matriz em setores retangulares espacialmente disjuntos, associados a subconjuntos do grafo. Essa abordagem modifica a granularidade da exploração do espaço de posicionamento, restringindo a busca a regiões físicas menores, sem impor isolamento topológico rígido entre os subconjuntos do grafo. As arestas entre as seções permanecem presentes e são consideradas durante o posicionamento, de modo que nós conectados a outros setores sejam posicionados preferencialmente próximos às bordas das regiões correspondentes.

Uma possibilidade de atribuição dos nós aos setores pode ser realizada por meio de um processo de agrupamento baseado na estrutura do grafo com o algoritmo *k-medoids*. Cada *cluster* do grafo pode ser associado a um setor físico da matriz de posicionamento, estabelecendo uma correspondência direta entre a estrutura lógica e a região espacial. Com isso, os algoritmos YOTO e YOTT passam a operar de forma local dentro de cada setor, reduzindo o número médio de tentativas por nó e mitigando os efeitos de congestionamento em cenários de alta densidade.

Outro eixo importante refere-se a trabalhos futuros para posicionadores em QCA com a execução dos algoritmos YOTO e YOTT sobre conjuntos completos de *benchmarks*. Até o momento, os experimentos conduzidos concentraram-se no algoritmo YOTT. Como trabalho futuro, pretende-se realizar uma avaliação comparativa entre os algoritmos SA, YOTO e YOTT, considerando diferentes esquemas de sincronização, com ênfase nas topologias baseadas em USE e *2DDWave*.

Os resultados mostram um *trade-off* entre a qualidade das soluções e o tempo de execução. Enquanto o algoritmo SA produz soluções de qualidade ao custo de um maior esforço computacional, os algoritmos baseados em travessia geram soluções próximas do ótimo com um tempo de execução reduzido, mostrando-se mais adequados para aplicações que demandam respostas rápidas ou mapeamento *on-the-fly*.

Referências Bibliográficas

[1] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. The epfl combinational benchmark suite. In *Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS)*, 2015.

[2] AMD. *UltraRAM Summary — Versal Adaptive SoC Memory Resources (AM007)*. Advanced Micro Devices, Inc., 2024. Accessed: 2025-06-24.

[3] AMD. *UltraScale Architecture and Product Data Sheet: Overview (DS890)*. Advanced Micro Devices, Inc., 2024. Accessed: 2025-06-24.

[4] AMD, Inc. Block ram introduction – versal adaptive soc memory resources. `https://docs.amd.com/r/en-US/am007-versal-memory/Block-RAM-Introduction`, 2024. Accessed: 2025-06-24.

[5] Jason Anderson, Rami Beidas, Vimal Chacko, Hsuan Hsiao, Xiaoyi Ling, Omar Ragheb, Xinyuan Wang, and Tianyi Yu. Cgra-me: An open-source framework for cgra architecture and cad research. In *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 156–162. IEEE, 2021.

[6] Marta Andronic and George A Constantinides. Polylut: learning piecewise polynomials for ultra-low latency fpga lut-based inference. In *2023 International Conference on Field Programmable Technology (ICFPT)*, pages 60–68. IEEE, 2023.

[7] Marta Andronic and George A Constantinides. Neuralut: Hiding neural network density in boolean synthesizable functions. In *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 140–148. IEEE, 2024.

[8] Marta Andronic and George A Constantinides. Neuralut-assemble: Hardware-aware assembling of sub-neural networks for efficient lut inference. In *2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 208–216. IEEE, 2025.

[9] Alan T. L. Bacellar, Zachary Susskind, Mauricio Breternitz Jr., Eugene John, Lizy K. John, Priscila M. V. Lima, and Felipe M. G. França. Differentiable weightless neural networks. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

[10] Rami Beidas and Jason H Anderson. Cgra mapping using zero-suppressed binary decision diagrams. In *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 616–622. IEEE, 2022.

[11] Vaughn Betz and Jonathan Rose. Vpr: A new packing, placement and routing tool for fpga research. In *International Workshop on Field Programmable Logic and Applications*, pages 213–222. Springer, 1997.

[12] Caio Araujo T Campos, Abner L Marciano, Omar P Vilela Neto, and Frank Sill Torres. Use: a universal, scalable, and efficient clocking scheme for qca. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 35(3):513–517, 2015.

[13] Raphael R Campos, Ricardo Ferreira, Julio C Goldner Vendramini, Fábio Cerqueira, and Marcelo Lobato Martins. Simulation of scale free gene regulatory networks based on threshold functions on gpu. In *Simpósio em Sistemas Computacionais de Alto Desempenho (SSCAD)*, pages 81–88. SBC, 2011.

[14] Michael Canesche. Algoritmos de posicionamento e roteamento baseados em travessia de grafo para arquiteturas reconfiguráveis de grão grosso (cgra). 2021.

[15] Michael Canesche, Westerley Carvalho, Lucas Reis, Matheus Oliveira, Salles Magalhães, Peter Jamieson, Jaugusto M Nacif, and Ricardo Ferreira. You only traverse twice: A yott placement, routing, and timing approach for cgras. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5s):1–25, 2021.

[16] Michael Canesche, Marcelo Menezes, Westerley Carvalho, Frank Sill Torres, Peter Jamieson, José Augusto Nacif, and Ricardo Ferreira. Traversal: A fast and adaptive graph-based placement and routing for cgras. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(8):1600–1612, 2020.

[17] Westerley Carvalho, Michael Canesche, Lucas Reis, Frank Torres, Lucas Silva, Peter Jamieson, José Nacif, and Ricardo Ferreira. A design exploration of scalable mesh-based fully pipelined accelerators. In *International Conference on Field-Programmable Technology (ICFPT)*, 2020.

[18] Oliver Cassidy, Marta Andronic, Samuel Coward, and George A Constantinides. Reducedlut: Table decomposition with"don't care"conditions. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 36–42, 2025.

[19] Chin-Chih Chang, Jason Cong, and Min Xie. Optimality and scalability study of existing placement algorithms. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, pages 621–627, 2003.

[20] Liang Chen and Tulika Mitra. Graph minor approach for application mapping on cgras. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(3):1–25, 2014.

[21] D-J Chyan and Melvin A Breuer. A placement algorithm for array processors. In *20th Design Automation Conference Proceedings*, pages 182–188. IEEE, 1983.

[22] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. Fpga hls today: successes, challenges, and opportunities. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 15(4):1–42, 2022.

[23] Jason Cong, Vivek Sarkar, Glenn Reinman, and Alex Bui. Customizable domain-specific computing. *IEEE Design & Test of Computers*, 28(2):6–15, 2010.

[24] Jeronimo Costa Penha, Lucas Bragança, Michael Canesche, Dener Ribeiro, José Augusto M Nacif, and Ricardo S Ferreira. Gene regulatory accelerators on cloud fpga. *Concurrency and Computation: Practice and Experience*, 35(24):e7822, 2023.

[25] Marcos Vinicius Da Silva, Ricardo Ferreira, Alisson Garcia, and Joao M. P. Cardoso. Mesh mapping exploration for coarse-grained reconfigurable array architectures. In *IEEE Int Conf on Reconfigurable Computing and FPGA's (ReConFig 2006)*, 2006.

[26] Shounak Dhar, Love Singhal, Mahesh Iyer, and David Pan. Fpga accelerated fpga placement. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 404–410. IEEE, 2019.

[27] Caleb Donovick, Makai Mann, Clark Barrett, and Pat Hanrahan. Agile SMT-based mapping for CGRAs with restricted routing networks. In *Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2019.

[28] Jorge Echavarria, Oliver Keszocze, and Jürgen Teich. Probability-based dse of approximated lut-based fpga designs. In *2022 IEEE 15th Dallas Circuit And System Conference (DCAS)*, pages 1–5. IEEE, 2022.

[29] Mohamed A Elgammal, Amin Mohaghegh, Soheil Gholami Shahrouz, Fatemehsadat Mahmoudi, Fahrican Koşar, Kimia Talaei, Joshua Fife, Daniel Khadivi, Kevin Murray, Andrew Boutros, et al. Vtr 9: Open-source cad for fabric and beyond fpga architecture exploration. *ACM Transactions on Reconfigurable Technology and Systems*, 18(3):1–53, 2025.

[30] Ricardo Ferreira, Vinicius Duarte, Monica Pereira, Luigi Carro, and Stephan Wong. A just-in-time modulo scheduling for virtual coarse-grained reconfig-

urable architectures. In *Int Conf on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2013.

[31] Ricardo Ferreira, Alisson Garcia, Tiago Teixeira, and Joao MP Cardoso. A polynomial placement algorithm for data driven coarse-grained reconfigurable architectures. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI'07)*, pages 61–66. IEEE, 2007.

[32] Ricardo Ferreira, Luciana Rocha, A Santos, J Nacif, Stephan Wong, and Luigi Carro. A run-time graph-based polynomial placement and routing algorithm for virtual fpgas. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–8. IEEE, 2013.

[33] Ricardo Ferreira, Luciana Rocha, Andre G Santos, Jose AM Nacif, Stephan Wong, and Luigi Carro. A runtime fpga placement and routing using low-complexity graph traversal. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 8(2):1–16, 2015.

[34] Ricardo Ferreira and Julio C Goldner Vendramini. Fpga-accelerated attractor computation of scale free gene regulatory networks. In *2010 International Conference on Field Programmable Logic and Applications*, pages 550–555. IEEE, 2010.

[35] Geraldo Fontes, Pedro Arthur RL Silva, Jose Augusto M Nacif, Omar P Vilela Neto, and Ricardo Ferreira. Placement and routing by overlapping and merging qca gates. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2018.

[36] Marcel Gort and Jason H Anderson. Analytical placement for heterogeneous fpgas. In *22nd international conference on field programmable logic and applications (FPL)*, pages 143–150. IEEE, 2012.

[37] Licheng Guo, Pongstorn Maidee, Yun Zhou, Chris Lavin, Jie Wang, Yuze Chi, Weikang Qiao, Alireza Kaviani, Zhiru Zhang, and Jason Cong. Rapidstream: Parallel physical implementation of fpga hls designs. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 1–12, 2022.

[38] Zeyu Guo. A survey on lut-based deep neural networks implemented in fpgas. *arXiv preprint arXiv:2506.07367*, 2025.

[39] Zeyu Guo. A survey on lut-based deep neural networks implemented in fpgas. *arXiv preprint arXiv:2506.07367*, 2025.

[40] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. Epimap: Using epimorphism to map applications on cgras. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1284–1291, 2012.

[41] Scott Hauck and Andre DeHon. *Reconfigurable computing: the theory and practice of FPGA-based computation*. Elsevier, 2010.

[42] Chengyu Hu, Qinghua Duan, Liran Hu, Peng Lu, Zhengjie Li, Meng Yang, Jian Wang, and Jinmei Lai. An analytical-based hybrid algorithm for fpga placement. In *Proceedings of the 2019 Great Lakes Symposium on VLSI*, pages 351–354, 2019.

[43] Vinícius Vilar Jacob, Chaulio de Resende Ferreira, and Ricardo Ferreira. Gpu optimization techniques applied to scale free gene regulatory networks based on threshold function. In *2012 13th Symposium on Computer Systems*, pages 57–64. IEEE, 2012.

[44] Alireza Khataei and Kia Bazargan. Treelut: An efficient alternative to deep neural networks for inference acceleration using gradient boosted decision trees. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 14–24, 2025.

[45] Jürgen M Kleinhans, Georg Sigl, and Frank M Johannes. Gordian: A new global optimization/rectangle dissection method for cell placement. In *The Best of ICCAD: 20 Years of Excellence in Computer-Aided Design*, pages 499–507. Springer, 1988.

[46] Takuya Kojima, Ayaka Ohwada, and Hideharu Amano. Mapping-aware kernel partitioning method for cgras assisted by deep learning. *IEEE Transactions on Parallel and Distributed Systems*, 33(5), 2021.

[47] Xiangyu Kong, Yi Huang, Jianfeng Zhu, Xingchen Man, Yang Liu, Chunyang Feng, Pengfei Gou, Minggui Tang, Shaojun Wei, and Leibo Liu. Mapzero: Mapping for coarse-grained reconfigurable architectures with reinforcement learning and monte-carlo tree search. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–14, 2023.

[48] Yen-Tai Lai, Hsin-Ya Lai, and Chia-Nan Yeh. Placement for the reconfigurable datapath architecture. In *2005 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1875–1878. IEEE, 2005.

[49] Craig S Lent, P Douglas Tougaw, Wolfgang Porod, and Gary H Bernstein. Quantum cellular automata. *Nanotechnology*, 4(1):49, 1993.

[50] Gaisheng Li, Fei Peng, Bing Zhang, Yanshuai Li, and Guangjun Xie. A qca placement and routing algorithm based on the sa algorithm. *International Journal of Electronics*, 111(12):2143–2158, 2024.

[51] Jiangnan Li, Chang Cai, Yaya Zhao, Yazhou Yan, Wenbo Yin, and Lingli Wang. Graft: Gnn-based adaptive framework for efficient cgra mapping. In *International Conference on Field Programmable Technology (ICFPT)*. IEEE, 2023.

[52] Wuxi Li, Yibo Lin, and David Z Pan. elfplace: Electrostatics-based placement for large-scale heterogeneous fpgas. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2019.

[53] Zhaoying Li, Dan Wu, Dhananjaya Wijerathne, and Tulika Mitra. Lisa: Graph neural network based portable mapping on spatial accelerators. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 444–459. IEEE, 2022.

[54] Dajiang Liu, Shouyi Yin, Guojie Luo, Jiaxing Shang, Leibo Liu, Shaojun Wei, Yong Feng, and Shangbo Zhou. Data-flow graph mapping optimization for CGRA with deep reinforcement learning. *IEEE Trans on Computer-Aided Design of Integrated Circuits and Systems*, 38(12), 2018.

[55] Binglei Lou, Richard Rademacher, David Boland, and Philip HW Leong. Polylut-add: Fpga-based lut inference with wide inputs. In *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 149–155. IEEE, 2024.

[56] Binglei Lou, Ruilin Wu, and Philip Leong. Sparselut: Sparse connectivity optimization for lookup table-based deep neural networks. In *2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 270–270. IEEE, 2025.

[57] Jason Luu, Ian Kuon, Peter Jamieson, Ted Campbell, Andy Ye, Wei Mark Fang, Kenneth Kent, and Jonathan Rose. Vpr 5.0: Fpga cad and architecture exploration tools with single-driver routing, heterogeneity and process scaling. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 4(4):1–23, 2011.

[58] Kevin JM Martin. Twenty years of automated methods for mapping applications on cgra. In *2022 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pages 679–686. IEEE, 2022.

[59] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. Dresc: A retargetable compiler for coarse-grained reconfigurable ar-

chitectures. In *2002 IEEE International Conference on Field-Programmable Technology, 2002.(FPT). Proceedings.*, pages 166–173. IEEE, 2002.

[60] Alan Mishchenko et al. Abc: A system for sequential synthesis and verification. *URL http://www. eecs. berkeley. edu/alanmi/abc*, 17, 2007.

[61] Seyed Mehdi Mohtavipour and Hadi Shahriar Shahhoseini. Gcn-ra: A graph convolutional network-based resource allocator for reconfigurable systems. *Journal of Computational Science*, 74:102178, 2023.

[62] Icaro Moreira, Lucas Bragança, Olavo Silva, Alysson Silva, Ricardo S Ferreira, and José Augusto M Nacif. Bridging the gap: Accelerating random forests on fpgas with high-bandwidth memory. In *2025 IEEE 16th Latin America Symposium on Circuits and Systems (LASCAS)*, volume 1, pages 1–5. IEEE, 2025.

[63] Chandra Mulpuri and Scott Hauck. Runtime and quality tradeoffs in FPGA placement and routing. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 29–36, 2001.

[64] Kevin E Murray, Oleg Petelin, Sheng Zhong, Jia Min Wang, Mohamed Eldafrawy, Jean-Philippe Legault, Eugene Sha, Aaron G Graham, Jean Wu, Matthew JP Walker, et al. VTR 8: High-performance CAD and Customizable FPGA architecture modelling. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 13(2):1–55, 2020.

[65] Tony Nowatzki, Newsha Ardalani, Karthikeyan Sankaralingam, and Jian Weng. Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pages 1–15, 2018.

[66] Jeronimo Penha, Alysson KC da Silva, Olavo Barros, Icaro Moreira, José Augusto M Nacif, and Ricardo Ferreira. Avaliação de estilos de código para árvores de decisão em gpu com microbenchmarks. In *Simpósio em Sistemas Computacionais de Alto Desempenho (SSCAD)*, pages 277–288. SBC, 2023.

[67] Jeronimo Costa Penha, Lucas Bragança da Silva, Michael Canesche, José Augusto M Nacif, and Ricardo Ferreira. Simulated annealing em hardware com multiplas threads em pipeline para posicionamento em cgras. In *Anais do XXIII Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 97–108. SBC, 2022.

[68] Rachel Selina Rajarathnam, Mohamed Baker Alawieh, Zixuan Jiang, Mahesh Iyer, and David Z Pan. Dreamplacefpga: An open-source analytical placer for large scale heterogeneous fpgas using deep-learning toolkit. In *2022 27th Asia*

and South Pacific Design Automation Conference (ASP-DAC), pages 300–306. IEEE, 2022.

[69] Rachel Selina Rajarathnam, Kate Thurmer, Vaughn Betz, Mahesh A Iyer, and David Z Pan. Better together: Combining analytical and annealing methods for fpga placement. In *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 43–52. IEEE, 2024.

[70] Charles Oliver Shields Jr. *Area efficient layouts of binary trees in grids*. The University of Texas at Dallas, 2001.

[71] Chang Sun, Zhiqiang Que, Vladimir Loncar, Wayne Luk, and Maria Spiropulu. da4ml: Distributed arithmetic for real-time neural networks on fpgas. *arXiv preprint arXiv:2507.04535*, 2025.

[72] Enrico Tabanelli, Giuseppe Tagliavini, and Luca Benini. Dnn is not all you need: Parallelizing non-neural ml algorithms on ultra-low-power iot processors. *ACM Transactions on Embedded Computing Systems*, 22(3):1–33, 2023.

[73] Shinya Takamaeda. A high-level hardware design environment in python. *IEICE Technical Report; IEICE Tech. Rep.*, 115(228):21–26, 2015.

[74] Ezra Thomas, Jing Li, and André DeHon. Hardware accelerated fpga divide-and-conquer page placement in milliseconds. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '26)*, Seaside, CA, USA, February 2026. ACM. To appear.

[75] Chunsheng Tian, Lei Chen, Yuan Wang, Shuo Wang, Jing Zhou, Yaowei Zhang, and Guang Li. Improving simulated annealing algorithm for fpga placement based on reinforcement learning. In *2022 IEEE 10th Joint International Information Technology and Artificial Intelligence Conference (ITAIC)*, volume 10, pages 1912–1919. IEEE, 2022.

[76] Cristian Tirelli, Juan Sapriza, Rubén Rodríguez Álvarez, Lorenzo Ferretti, Benoît Denkinger, Giovanni Ansaloni, José Miranda Calero, David Atienza, and Laura Pozzi. Sat-based exact modulo scheduling mapping for resource-constrained cgras. *arXiv preprint arXiv:2402.12834*, 2024.

[77] Aashish Kumar Tiwary, Saketh Gajawada, Jay Shah, and Nanditha Rao. Lutaccel: Look-up-table based vector systolic accelerator on fpgas. In *2025 26th International Symposium on Quality Electronic Design (ISQED)*, pages 1–7. IEEE, 2025.

[78] Frank Sill Torres, Pedro A Silva, Geraldo Fontes, Marcel Walter, José Augusto M Nacif, Ricardo Santos Ferreira, Omar Paranaiba Vilela Neto, Jeferson F Chaves,

Robert Wille, Philipp Niemann, et al. On the impact of the synchronization constraint and interconnections in quantum-dot cellular automata. *Microprocessors and Microsystems*, 76:103109, 2020.

[79] Frank Sill Torres, Robert Wille, Marcel Walter, Philipp Niemann, Daniel Große, and Rolf Drechsler. Evaluating the impact of interconnections in quantum-dot cellular automata. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 649–656. IEEE, 2018.

[80] P Douglas Tougaw and Craig S Lent. Logical devices implemented using quantum cellular automata. *Journal of Applied physics*, 75(3):1818–1825, 1994.

[81] Verilog-to-Routing Project. Vpr: File formats, 2025. Accessed: 7 June 2025.

[82] Maria Vieira, Michael Canesche, Lucas Bragança, Josué Campos, Mateus Silva, Ricardo Ferreira, and Jose A Nacif. Reshape: A run-time dataflow hardware-based mapping for cgra overlays. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2021.

[83] Kizheppatt Vipin and Suhaib A Fahmy. An approach to a fully automated partial reconfiguration design flow. In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 231–231. IEEE, 2013.

[84] Matthew JP Walker and Jason H Anderson. Generic connectivity-based CGRA mapping via integer linear programming. In *Field-Programmable Custom Computing Machines (FCCM)*, 2019.

[85] Matthew JP Walker and Jason H Anderson. Generic connectivity-based cgra mapping via integer linear programming. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 65–73. IEEE, 2019.

[86] Marcel Walter, Benjamin Hien, and Robert Wille. Versatile signal distribution networks for scalable placement and routing of field-coupled nanocomputing technologies. In *2023 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 1–6. IEEE, 2023.

[87] Marcel Walter and Robert Wille. Efficient multi-path signal routing for field-coupled nanotechnologies. In *Proceedings of the 17th ACM International Symposium on Nanoscale Architectures*, pages 1–6, 2022.

[88] Marcel Walter, Robert Wille, Daniel Große, Frank Sill Torres, and Rolf Drechsler. An exact method for design exploration of quantum-dot cellular automata. In

2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 503–508. IEEE, 2018.

[89] Marcel Walter, Robert Wille, Daniel Große, Frank Sill Torres, and Rolf Drechsler. Placement and routing for tile-based field-coupled nanocomputing circuits is np-complete (research note). *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 15(3):1–10, 2019.

[90] Chris C Wang and Guy GF Lemieux. Scalable and deterministic timing-driven parallel placement for FPGAs. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 153–162, 2011.

[91] Shang Wang, Deepak Ranganatha Sastry Mamillapalli, Qianxi Li, Tianpei Yang, and Matthew E Taylor. Reinforcement learning for fpga placement. In *Machine Learning for Systems Workshop at 37th NeurIPS Conference*, New Orleans, LA, USA, December 2023.

[92] Xusheng Wang. Designing digital circuits based on quantum-dots cellular automata using nature-inspired metaheuristic algorithms: A systematic literature review. *Optik*, 262:169251, 2022.

[93] Jian Weng, Sihao Liu, Dylan Kupsh, and Tony Nowatzki. Unifying spatial accelerator compilation with idiomatic and modular transformations. *IEEE Micro*, (01):1–12, 2022.

[94] Olivia Weng, Marta Andronic, Danial Zuberi, Jiaqing Chen, Caleb Geniesse, George A Constantinides, Nhan Tran, Nicholas J Fraser, Javier Mauricio Duarte, and Ryan Kastner. Greater than the sum of its luts: Scaling up lut-based neural networks with amigolut. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 25–35, 2025.

[95] Dhananjaya Wijerathne, Zhaoying Li, Thilini Kaushalya Bandara, and Tulika Mitra. Panorama: divide-and-conquer approach for mapping complex loop kernels on cgra. In *ACM/IEEE Design Automation Conference*, pages 127–132, 2022.

[96] Henry Wong, Vaughn Betz, and Jonathan Rose. Quantifying the gap between fpga and custom cmos to aid microarchitectural design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(10):2067–2080, 2013.

[97] Michael G Wrighton and André M DeHon. Hardware-assisted simulated annealing with application for fast FPGA placement. In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 33–42, 2003.

[98] Yuanlong Xiao, Eric Micallef, Andrew Butt, Matthew Hofmann, Marc Alston, Matthew Goldsmith, Andrew Merczynski-Hait, and André DeHon. Pld: Fast fpga compilation to make reconfigurable acceleration compatible with modern incremental refinement software development. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 933–945, 2022.

[99] DS890 Xilinx. Ultrascale architecture and product data sheet: Overview. *Product Specification, Nov*, 2020.

[100] Yonghong Xu and Mohammed AS Khalid. Qpf: efficient quadratic placement for fpgas. In *International Conference on Field Programmable Logic and Applications, 2005.*, pages 555–558. IEEE, 2005.

[101] Jonghee W Yoon, Aviral Shrivastava, Sanghyun Park, Minwook Ahn, and Yunheung Paek. A graph drawing based spatial mapping algorithm for coarse-grained reconfigurable architectures. *IEEE transactions on very large scale integration (VLSI) systems*, 17(11):1565–1578, 2009.

[102] Yan Zhuang, Zhihao Zhang, and Dajiang Liu. Towards high-quality cgra mapping with graph neural networks and reinforcement learning. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9, 2022.

Apêndice A

Exemplo de descrição de arquitetura XML LUT-4 para VPR 5.0.2

```
<architecture>
    <layout auto="1.000000"/>
    <device>
        <sizing R_minW_nmos="5726.870117" R_minW_pmos="15491.700195
           " ipin_mux_trans_size="1.000000"/>
        <timing C_ipin_cblock="1.191000e-14" T_ipin_cblock="
           1.482000e-10"/>
        <area grid_logic_tile_area="30000.000000"/>
        <chan_width_distr>
           <io width="1.000000"/>
           <x distr="uniform" peak="1.000000"/>
           <y distr="uniform" peak="1.000000"/>
        </chan_width_distr>
        <switch_block type="wilton" fs="3"/>
    </device>
    <switchlist>
        <switch type="mux" name="0" R="94.841003" Cin="1.537000e-14
           " Cout="2.194000e-13" Tdel="6.562000e-11" mux_trans_size
           ="10.000000" buf_size="1"/>
    </switchlist>
    <segmentlist>
        <segment freq="1.000000" length="1" type="unidir" Rmetal="
           11.064550" Cmetal="4.727860e-14">
           <mux name="0"/>
           <sb type="pattern">1 1</sb>
           <cb type="pattern">1</cb>
        </segment>
    </segmentlist>


```

```
    <typelist>
        <io capacity="3" t_inpad="7.734000e-11" t_outpad="4.395000e
           -11">
           <fc_in type="frac">0.250000</fc_in>
           <fc_out type="frac">1</fc_out>
        </io>
        <type name=".clb">
            <subblocks max_subblocks="4" max_subblock_inputs="4">
                <timing>
                    <T_comb>
                        <trow>1.679000e-10</trow>
                        <trow>1.679000e-10</trow>
                        <trow>1.679000e-10</trow>
                        <trow>1.679000e-10</trow>
                    </T_comb>
                    <T_seq_in>
                        <trow>-3.990000e-11</trow>
                    </T_seq_in>
                    <T_seq_out>
                        <trow>1.261000e-10</trow>
                    </T_seq_out>
                </timing>
            </subblocks>
            <fc_in type="frac">0.250000</fc_in>
            <fc_out type="frac">1</fc_out>
            <pinclasses>
                <class type="in">0 1 2 3 4 5 6 7 8 9 </class>
                <class type="out">10 11 12 13 </class>
                <class type="global">14 </class>
            </pinclasses>
            <pinlocations>
                <loc side="left">1 5 9 11 </loc>
                <loc side="right">3 7 13 </loc>
                <loc side="top">2 6 12 14 </loc>
                <loc side="bottom">0 4 8 10 </loc>
            </pinlocations>
            <gridlocations>
                <loc type="fill" priority="1"/>
            </gridlocations>

```

```
66              <timing>
67                  <tedge type="T_sblk_opin_to_sblk_ipin">1.042000e-10
                        </tedge>
68                  <tedge type="T_fb_ipin_to_sblk_ipin">9.955000e-11</
                        tedge>
69                  <tedge type="T_sblk_opin_to_fb_opin">0.000000e+00</
                        tedge>
70              </timing>
71          </type>
72      </typelist>
73  </architecture>
```

Listing A.1: Arquitetura LUT-4 utilizado nos experimentos da Seção 4.2.1

Apêndice B

Exemplo de descrição de arquitetura XML LUT-8 para VPR 9.0.0

```
<architecture>
    <models/>
    <tiles>
        <tile name="io" area="0">
            <sub_tile name="io" capacity="8">
                <equivalent_sites>
                    <site pb_type="io" pin_mapping="direct"/>
                </equivalent_sites>
                <input name="outpad" num_pins="1"/>
                <output name="inpad" num_pins="1"/>
                <clock name="clock" num_pins="1"/>
                <fc in_type="frac" in_val="0.15" out_type="frac"
                   out_val="0.15"/>
                <pinlocations pattern="custom">
                    <loc side="left">io.outpad io.inpad io.clock</
                       loc>
                    <loc side="top">io.outpad io.inpad io.clock</
                       loc>
                    <loc side="right">io.outpad io.inpad io.clock</
                       loc>
                    <loc side="bottom">io.outpad io.inpad io.clock<
                       /loc>
                </pinlocations>
            </sub_tile>
        </tile>
        <tile name="clb" area="18000">
            <sub_tile name="clb">
                <equivalent_sites>
                    <site pb_type="clb" pin_mapping="direct"/>
                </equivalent_sites>
```

```
26                  <input name="I" num_pins="6"/>
27                  <output name="O" num_pins="1"/>
28                  <clock name="clk" num_pins="1"/>
29                  <fc in_type="frac" in_val="0.15" out_type="frac"
                       out_val="0.15"/>
30                  <pinlocations pattern="spread"/>
31              </sub_tile>
32          </tile>
33      </tiles>
34      <layout>
35          <auto_layout aspect_ratio="1.0">
36              <perimeter type="io" priority="100"/>
37              <corners type="EMPTY" priority="101"/>
38              <fill type="clb" priority="10"/>
39          </auto_layout>
40      </layout>
41      <device>
42          <sizing R_minW_nmos="8926" R_minW_pmos="16067"/>
43          <area grid_logic_tile_area="0"/>
44          <chan_width_distr>
45              <x distr="uniform" peak="1.0"/>
46              <y distr="uniform" peak="1.0"/>
47          </chan_width_distr>
48          <switch_block type="wilton" fs="3"/>
49          <connection_block input_switch_name="ipin_cblock"/>
50      </device>
51      <switchlist>
52          <switch type="mux" name="0" R="551" Cin=".77e-15" Cout="4e
               -15" Tdel="58e-12"
53                  mux_trans_size="2.630740" buf_size="27.645901"/>
54          <switch type="mux" name="ipin_cblock" R="2231.5" Cout="0."
               Cin="1.47e-15"
55                  Tdel="7.247000e-11" mux_trans_size="1.222260"
                       buf_size="auto"/>
56      </switchlist>
57      <segmentlist>
58          <segment freq="1.0" length="4" type="unidir" Rmetal="101"
               Cmetal="22.5e-15">
59              <mux name="0"/>
60              <sb type="pattern">1 1 1 1 1</sb>
61              <cb type="pattern">1 1 1 1</cb>
```

```
62          </segment>
63      </segmentlist>
64      <complexblocklist>
65          <pb_type name="io">
66              <input name="outpad" num_pins="1"/>
67              <output name="inpad" num_pins="1"/>
68              <clock name="clock" num_pins="1"/>
69              <mode name="inpad">
70                  <pb_type name="inpad" blif_model=".input" num_pb="1
                        ">
71                      <output name="inpad" num_pins="1"/>
72                  </pb_type>
73                  <interconnect>
74                      <direct name="inpad" input="inpad.inpad" output
                            ="io.inpad">
75                          <delay_constant max="4.243e-11" in_port="
                                inpad.inpad" out_port="io.inpad"/>
76                      </direct>
77                  </interconnect>
78              </mode>
79              <mode name="outpad">
80                  <pb_type name="outpad" blif_model=".output" num_pb=
                        "1">
81                      <input name="outpad" num_pins="1"/>
82                  </pb_type>
83                  <interconnect>
84                      <direct name="outpad" input="io.outpad" output=
                            "outpad.outpad">
85                          <delay_constant max="1.394e-11" in_port="io
                                .outpad" out_port="outpad.outpad"/>
86                      </direct>
87                  </interconnect>
88              </mode>
89              <power method="ignore"/>
90          </pb_type>
91          <pb_type name="clb">
92              <input name="I" num_pins="6"/>
93              <output name="O" num_pins="1"/>
94              <clock name="clk" num_pins="1"/>
95              <pb_type name="lut6" blif_model=".names" num_pb="1"
                    class="lut">
```

```
96                  <input name="in" num_pins="6" port_class="lut_in"/>
97                  <output name="out" num_pins="1" port_class="lut_out
                        "/>
98                  <delay_matrix type="max" in_port="lut6.in" out_port
                        ="lut6.out">
99                      82e-12
100                     173e-12
101                     261e-12
102                     263e-12
103                     398e-12
104                     397e-12
105                 </delay_matrix>
106             </pb_type>
107             <interconnect>
108                 <direct name="inputs" input="clb.I" output="lut6.in
                        "/>
109                 <direct name="outputs" input="lut6.out" output="clb
                        .O"/>
110             </interconnect>
111         </pb_type>
112     </complexblocklist>
113     <power>
114         <local_interconnect C_wire="2.5e-10"/>
115         <mux_transistor_size mux_transistor_size="3"/>
116         <FF_size FF_size="4"/>
117         <LUT_transistor_size LUT_transistor_size="4"/>
118     </power>
119     <clocks>
120         <clock buffer_size="auto" C_wire="2.5e-10"/>
121     </clocks>
122 </architecture>
```

Listing B.1: Arquitetura LUT-8 utilizado nos experimentos da Seção 4.2.1

Apêndice C

Exemplo dos arquivos BLIF e PLACE para o VPR nas versões 5 e 9

```
.clb 0
pinlist: 5 8 16 19 open open 0 open
subblock: 0 0 1 2 3 open open 6 open

.clb 1
pinlist: 5 8 9 19 open open 1 open
subblock: 1 0 1 2 3 open open 6 open

.clb 2
pinlist: 8 9 16 19 28 open 2 open
subblock: 2 0 1 2 3 4 open 6 open

.clb 3
pinlist: 5 8 16 19 27 open 3 open
subblock: 3 0 1 2 3 4 open 6 open

.clb 4
pinlist: 8 9 16 19 open open 4 open
subblock: 4 0 1 2 3 open open 6 open

.input 5
pinlist: 5

.clb 6
pinlist: 8 9 16 19 open open 6 open
subblock: 6 0 1 2 3 open open 6 open

.clb 7
pinlist: 8 9 16 19 open open 7 open
subblock: 7 0 1 2 3 open open 6 open
```

```
31
32 .input 8
33 pinlist: 8
34
35 .input 9
36 pinlist: 9
37
38 .clb 10
39 pinlist: 8 9 16 19 open open 10 open
40 subblock: 10 0 1 2 3 open open 6 open
41
42 .clb 11
43 pinlist: 8 9 16 19 open open 11 open
44 subblock: 11 0 1 2 3 open open 6 open
45
46 .clb 12
47 pinlist: 5 8 16 19 open open 12 open
48 subblock: 12 0 1 2 3 open open 6 open
49
50 .clb 13
51 pinlist: 8 9 16 19 open open 13 open
52 subblock: 13 0 1 2 3 open open 6 open
53
54 .clb 14
55 pinlist: 8 9 16 19 open open 14 open
56 subblock: 14 0 1 2 3 open open 6 open
57
58 .clb 15
59 pinlist: 8 9 16 19 open open 15 open
60 subblock: 15 0 1 2 3 open open 6 open
61
62 .input 16
63 pinlist: 16
64
65 .clb 17
66 pinlist: 8 16 open open open open 17 open
67 subblock: 17 0 1 open open open open 6 open
68
69 .clb 18
70 pinlist: 8 9 16 19 open open 18 open
71 subblock: 18 0 1 2 3 open open 6 open
```

```
72
73 .input 19
74 pinlist: 19
75
76 .clb 20
77 pinlist: 8 9 16 19 open open 20 open
78 subblock: 20 0 1 2 3 open open 6 open
79
80 .clb 21
81 pinlist: 8 9 16 19 open open 21 open
82 subblock: 21 0 1 2 3 open open 6 open
83
84 .clb 22
85 pinlist: 3 16 open open open open 22 open
86 subblock: 22 0 1 open open open open 6 open
87
88 .clb 23
89 pinlist: 8 9 16 19 open open 23 open
90 subblock: 23 0 1 2 3 open open 6 open
91
92 .clb 24
93 pinlist: 5 8 16 25 open open 24 open
94 subblock: 24 0 1 2 3 open open 6 open
95
96 .clb 25
97 pinlist: 5 8 9 19 27 open 25 open
98 subblock: 25 0 1 2 3 4 open 6 open
99
100 .output out_26:5
101 pinlist: 5
102
103 .input 27
104 pinlist: 27
105
106 .clb 28
107 pinlist: 5 8 9 16 19 open 28 open
108 subblock: 28 0 1 2 3 4 open 6 open
109
110 .output out_29:17
111 pinlist: 17
112
```

```
.clb 30
pinlist: 8 9 16 19 open open 30 open
subblock: 30 0 1 2 3 open open 6 open

.clb 31
pinlist: 8 9 16 19 open open 31 open
subblock: 31 0 1 2 3 open open 6 open

.clb 32
pinlist: 5 9 16 19 open open 32 open
subblock: 32 0 1 2 3 open open 6 open

.output out_33:0
pinlist: 0

.output out_34:1
pinlist: 1

.output out_35:2
pinlist: 2

.output out_36:4
pinlist: 4

.output out_37:6
pinlist: 6

.output out_38:7
pinlist: 7

.output out_39:10
pinlist: 10

.output out_40:11
pinlist: 11

.output out_41:12
pinlist: 12

.output out_42:13
pinlist: 13
```

```
154
155 .output out_43:14
156 pinlist: 14
157
158 .output out_44:15
159 pinlist: 15
160
161 .output out_45:18
162 pinlist: 18
163
164 .output out_46:20
165 pinlist: 20
166
167 .output out_47:21
168 pinlist: 21
169
170 .output out_48:22
171 pinlist: 22
172
173 .output out_49:23
174 pinlist: 23
175
176 .output out_50:24
177 pinlist: 24
178
179 .output out_51:30
180 pinlist: 30
181
182 .output out_52:31
183 pinlist: 31
184
185 .output out_53:32
186 pinlist: 32
```

Listing C.1: Exemplo de arquivo BLIF para o *benchmark* **CTRL** citado na Seção 4.2.1

```
Netlist file: reports/fpga/EPFL/sa/net/ctrl_k6_0.net Architecture file: arch/k6-n1.xml
Array size: 8 x 8 logic blocks
#block name     X       Y       subblk  block_number

#----------     --      --      ------  ------------
0       3       2       0       #0
1       1       1       0       #1
2       2       3       0       #2
3       2       2       0       #3
4       3       5       0       #4
5       3       0       0       #5
6       2       4       0       #6
7       1       7       0       #7
8       0       3       0       #8
9       0       6       0       #9
10      1       8       0       #10
11      1       5       0       #11
12      3       3       0       #12
13      3       6       0       #13
14      1       6       0       #14
15      2       6       0       #15
16      0       4       0       #16
17      4       3       0       #17
18      3       4       0       #18
19      0       5       0       #19
20      1       2       0       #20
21      4       5       0       #21
22      4       2       0       #22
23      2       5       0       #23
24      3       1       0       #24
25      2       1       0       #25
out_26:5        7       0       0       #26
27      2       0       0       #27
28      1       3       0       #28
out_29:17       6       0       0       #29
30      2       7       0       #30
31      1       4       0       #31
32      4       4       0       #32
out_33:0        9       1       0       #33
out_34:1        1       0       0       #34
```

```
41 out_35:2          9          3          0          #35
42 out_36:4          3          9          0          #36
43 out_37:6          9          4          0          #37
44 out_38:7          0          8          0          #38
45 out_39:10         1          9          0          #39
46 out_40:11         7          9          0          #40
47 out_41:12         4          0          0          #41
48 out_42:13         4          9          0          #42
49 out_43:14         0          7          0          #43
50 out_44:15         2          9          0          #44
51 out_45:18         5          9          0          #45
52 out_46:20         0          1          0          #46
53 out_47:21         9          6          0          #47
54 out_48:22         9          2          0          #48
55 out_49:23         9          5          0          #49
56 out_50:24         5          0          0          #50
57 out_51:30         9          7          0          #51
58 out_52:31         0          2          0          #52
59 out_53:32         6          9          0          #53
```

Listing C.2: Exemplo de arquivo PLACE para o *benchmark* **CTRL** citado na Seção 4.2.1