MICHAEL CANESCHE

ALGORITMOS DE POSICIONAMENTO E ROTEAMENTO BASEADOS EM TRAVESSIA DE GRAFO PARA ARQUITETURAS RECONFIGURÁVEIS DE GRÃO GROSSO (CGRA)

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

Orientador: Ricardo dos Santos Ferreira

VIÇOSA - MINAS GERAIS
2021

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

T
C221a
2021

Canesche, Michael, 1991-
Algoritmos de posicionamento e roteamento baseados em travessia de grafo para arquiteturas reconfiguráveis de grão grosso (CGRA) / Michael Canesche. – Viçosa, MG, 2021.
107 f. : il. (algumas color.) ; 29 cm.

Orientador: Ricardo dos Santos Ferreira.
Dissertação (mestrado) - Universidade Federal de Viçosa.
Referências bibliográficas: f.96-107.

1. Arquitetura de computador. 2. Dispositivo de lógica programável. I. Universidade Federal de Viçosa. Departamento de Informática. Programa de Pós-Graduação em Ciência da Computação. II. Título.

CDD 22. ed. 004.22

MICHAEL CANESCHE

ALGORITMOS DE POSICIONAMENTO E ROTEAMENTO BASEADOS EM TRAVESSIA DE GRAFO PARA ARQUITETURAS RECONFIGURÁVEIS DE GRÃO GROSSO (CGRA)

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

APROVADA: 19 de fevereiro de 2021.

Assentimento:

Michael Canesche

Michael Canesche
Autor

Ricardo dos Santos Ferreira
Orientador

Dedico este trabalho aos meus pais e amigos.

AGRADECIMENTOS

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001.
Aos meus pais Miguel e Maria pelo total apoio e carinho. Ao meu irmão Miguel pelo companheirismo. Aos meus amigos por sempre estarem ao meu lado nessa longa caminhada.
À todos meus professores por me guiarem pelo caminho do conhecimento.
À Universidade Federal de Viçosa por todas as oportunidades.
Ao Professor Ricardo dos Santos Ferreira pela orientação, confiança, disponibilidade, paciência e apreço pela ciência.
À todos que diretamente ou indiretamente contribuíram com este trabalho.
Agradeço a FAPEMIG com o apoio através do projeto "Aceleradores para Redes Reguladores de Genes" - Modalidade: "Edital 001/2018 - Demanda Universal", Processo N. : APQ-01203-18.

It's the questions we can't answer that teach us the most. They teach us how to think. If you give a man an answer, all he gains is a little fact. But give him a question and he'll look for his own answers - P. Rothfuss, The Wise Man's Fear.

RESUMO

CANESCHE, Michael, M.Sc., Universidade Federal de Viçosa, fevereiro de 2021. **Algoritmos de Posicionamento e Roteamento baseados em Travessia de Grafo para Arquiteturas Reconfiguráveis de Grão Grosso (CGRA)**. Orientador: Ricardo dos Santos Ferreira.

A desaceleração da lei de Moore e o fim da escalabilidade de Dennard, criou novos desafios para prover desempenho, flexibilidade e eficiência energética na área de arquitetura de computadores. Processadores de propósito geral (CPUs), unidades de processamento gráfico (GPUs) e processadores de sinais digitais (DSPs) oferecem flexibilidade, mas possuem baixa eficiência energética. Circuitos integrados de aplicação específicas (ASICs) possuem uma alta eficiência energética, mas uma baixa flexibilidade. Arranjos de portas programáveis (FPGAs) tem eficiência e flexibilidade, entretanto requerem a necessidade de conhecimento especializado em *hardware* e em linguagens de descrição e o tempo de compilação leva de minutos a horas. Por outro lado, arquiteturas reconfiguráveis de grão grosso (CGRAs) apresentam flexibilidade próxima a um FPGA e sua eficiência energética é próxima a um ASIC. Entretanto, ainda faltam ferramentas para a popularização dos CGRAs, dentre elas podemos citar os algoritmos de posicionamento e roteamento (P&R). Esta dissertação apresenta dois algoritmos de P&R baseados em travessia de grafos para CGRA. O primeiro algoritmo introduz um novo percurso em Zigzag que explora a correlação entre as saídas, uma implementação em GPU e uma exploração sistemática do espaço de soluções. Os resultados mostraram uma aceleração de até 502x sem perda de qualidade em comparação ao estado da arte. O segundo algoritmo propõe uma nova abordagem de travessia capaz de explorar as reconvergências presentes nos grafos. Os resultados mostraram que a solução reduziu o comprimento dos fios em 7%, o tamanho máximo das filas em 2 vezes e melhorou o tempo de execução em até 4 vezes comparado com o primeiro algoritmo.

Palavras-chave: Arquiteturas Reconfiguráveis. CGRAs. Posicionamento. Roteamento.

ABSTRACT

CANESCHE, Michael, M.Sc., Universidade Federal de Viçosa, February, 2021. **Placement and Routing graph transversal-based Algorithms for Coarse-Grained Reconfigurable Architectures (CGRA)**. Advisor: Ricardo dos Santos Ferreira.

The slowdown in Moore's law and the end of Dennard's scalability has created new challenges in providing performance, flexibility, and energy efficiency in the area of computer architecture. General-purpose processors (CPUs), graphics processing units (GPUs), and digital signal processors (DSPs) offer flexibility but are low in energy efficiency. Application-specific integrated circuits (ASICs) have high energy efficiency but low flexibility. Field-programmable gate array (FPGAs) are flexible and energy efficient, but they require specialized knowledge in hardware and description language, not to mention that the compilation time takes from minutes to hours. On the other hand, reconfigurable coarse-grained architectures (CGRAs) have flexibility close to that of a FPGA, and their energy efficiency is comparable to that of an ASIC. However, tools for popularizing CGRAs are still lacking. Among them, we can mention the placement and routing algorithms (P&R). This dissertation presents two P&R algorithms based on graph crossing for CGRA. The first algorithm introduces a new path in Zigzag that explores the correlation between outputs, a GPU implementation, and a systematic exploration of the solution space. The results showed an acceleration of up to 502x without loss of quality than the state of the art. The second algorithm proposes a new crossing approach capable of exploring the reconvergences present in the graphs. The results shows that the solution reduced the wire length by 7%, the maximum queue size by 2x, and improved the execution time by up to 4 times compared to the first algorithm.

Keywords: Reconfigurable architecture. CGRAs. Placement. Routing.

LISTA DE FIGURAS

1.1 Qualidade da solução em função do tempo de escalonamento: (a) Cenário com uma Arquitetura com custo adicional de 50% e FIFOS elásticas de tamanho 15; (b) Cenário com uma arquitetura com FIFO elásticas de Tamanho 3 e custo adicional de 10%. Figura extraída e adaptada de Nowatzki et al. (2018a) 16
1.2 (a) Grafo de fluxo de dados; (b) P&R do grafo. 18

2.1 Estrutura básica de um CGRA bi-dimensional conectado entre os seus vizinhos de tamanho 4×4. Adaptado de (Kim et al., 2011b). 22
2.2 Topologias e interconexões: (a) linear; (b) malha; (c) 1-hop; 23
2.3 Topologias e interconexões mais complexas: (a) cluster; (b) strip; (c) multistágio; (d) cubo. 23
2.4 Arquiteturas PE: (a) dinâmico; (b) estático. 24
2.5 Mapeamento em um CGRA: (a) grafo; (b) mapeamento estático; (c) mapeamento dinâmico; (d) Modulo Scheduling. 25
2.6 Grafo MRRG para uma arquitetura em malha: (a) arquitetura 2×2 com duas partições temporais; (b) MRRG com II=2. 27
2.7 Mapeamento dinâmico: (a) Grafo de exemplo de entrada; (b) Mapeado II=2 no MRRG; (c) Arquitetura em malha 3×3; (d) Mapeamento estático. 27
2.8 Mapeamento dinâmico: (a) Grafo de entrada utilizando II=2; (b) Grafo de entrada utilizando II=3; (c) MRRG com II=3; (d) Mapeamento estático em uma arquitetura 3×3 em malha. 28
2.9 Exemplo de Mapeamento: (a) Grafo de entrada; (b) Solução inválida; (c) Solução válida com o uso de uma FIFO. 29
2.10 Técnicas de balanceamento: (a) Mapeamento ótimo; (b) Caminhos longos; (c) Uso da UP; (d) Filas ajustáveis. 29

3.1 Exemplo de mapeamento de grafo por níveis em uma arquitetura hexadecimal. Retirada de (Koren et al., 1988). 32
3.2 Mapeamento utilizando abordagem gulosa. (a) Grafo de entrada; (b) Mapeamento em um grafo por nível e profundidade; (c) Ajustes no mapeamento; (d) Roteamento do grafo. Adaptado de (Silva et al., 2006). 33
3.3 Compactação de níveis utilizando *Simulated Annealing*. (a) Grafo antes da compactação; (b) Grafo após a compactação. Adaptado de (Silva et al., 2006). 33
3.4 Percusso dos grafos. (a) Grafo de fluxo de dados; (b) Busca em profundidade; (c) Busca em amplitude. Adaptado de (Silva et al., 2006). 34
3.5 Estudo de caso do *Benchmark fdtd-apml*. (a) Posicionamento inicial; (b) Resultado do mapeamento após o primeiro passo; (c) Resultado final do mapeamento. Retirado de (Liu et al., 2018). 36
3.6 (a) Grafo de fluxo de dados; Tomadas de decisão: (b) Nodo inicial; (c) Posicionamento de um novo nodo; (d) Escolha de um novo nodo. 37

3.7 Comparação do Percurso em Profundidade com o Percurso ZigZag: (a) Grafo de Entrada com Travessia em Profundidade; (b) Grafo da Arquitetura em Profundidade; (c) Travessia em ZigZag do Grafo de Entrada; (d) Grafo da Arquitetura percorrido em ZigZag. 39
3.8 Reconvergência em grafos de fluxo de dados: (a) Percurso em Profundidade; (b) Uma solução no grafo da arquitetura em profundidade; (c) Percurso em Zigzag; (d) Mapeamento na arquitetura alvo para o percurso em Zigzag. 40
3.9 Algoritmo YOTT: (a) Grafo com anotações; (b) Posicionamento de F seguindo as anotações; (c) Posicionamento de E; (d) Solução ótima resultante das anotações do YOTT. 41

4.1 (a) A $4x4$ mesh-based CGRA; (b) 1-hop connection; (c) Short distance; (d) ADRES CGRA 1-hop plus one memory per row. 45
4.2 (a) A dataflow graph; (b) P&R with unbalanced highlighted wires; (c) Unbalanced due Routing Registers; (d) Balanced P&R. 45
4.3 (a) A latency= 4 and Initiation Interval= 4; (b) Initiation Interval= 2; (c) Modulo Scheduling $II = 2$, and two overlapping iterations. 46
4.4 (a) Unbalanced P&R; (b) FIFO Balanced P&R; (c) Optimal P&R; (d) Balanced P&R without FIFOs. 49
4.5 (a) P&R Pseudo-code (b) P&R Greedy Instances Space Parameters. . . . 50
4.6 (a) Traversal list L and an initial position for d; (b) Place c; (c) Place b and a; (d) Final P&R; (e) New initial position for d; (f) A feasible P&R. . 51
4.7 (a) Two Outputs; (b) Without correlated outputs; (c) ZigZag traversal. . 53
4.8 ZigZag Traversal: (a) First path; (b) Reverse point and second path; (c) Third path and reverse (node 8); (d) When add the edge $9 \rightarrow 2$, we detect a cycle $(2, 3, \ldots, 8, 9, 2)$, which starts and ends at node 2. Then, we pop the node 8 from the stack to continue the traversal by visiting node 10. 54
4.9 (a) An ADJ list as a relative distance vector; (b) Place node c; (c) A distinct Adj list generates a new placement of c. 55
4.10 (a) 1-hop and placement of b from the edge $b \rightarrow c$; (b) A new ADJ list; (c) Diagonal Interconnections and its ADJ list. 56
4.11 Modulo Scheduling: (a) Mesh matrix plus temporal dimension;(b) Initial position; (c-g) Placement sequence; (h) Feasible solution. 57
4.12 (a) Detailed view of elastic CGRA with FIFO size=2; (b) FIFO size implementation; (c) Symbolic view; (d) $LFS = 3$; (e) Splitting the LFS to reduce it to 1; (f) $LFS = 2$; (g) $LFS = 3$ which is reduced to 2 in (h). . . . 58
4.13 (a) Scan list pseudo-code; (b) $Thread_0$ finished first; (c-d) An example of execution sequence for two threads. 60
4.14 Execution for six strategies in 1-hop. Lower is better (log scale). 66
4.15 Fifo size for six strategies in 1-hop. Lower is better and zero is optimal. 66
4.16 Fifo size for six strategies in Mesh topology. Lower is better and zero is optimal. 66

5.1 (a) Execution time for SA and traversal placements in log scale; (b) Number of swaps for SA approaches and number of evaluation placement for traversal approaches. Lower is better. 72

5.2 (a-c) Percentage of optimal mapped edges for SA and traversal placements, higher is better; (d-f) Normalized maximum FIFO size for SA and traversal approaches, lower is better. 73
5.3 (a) Input dataflow graph and traversal decisions; Output Graph: (b) Start node; (c) Start position; (d-f) Neighborhood placement decision. . 74
5.4 (a) An two output data-flow; (b-g) A depth-first traversal placement step by step. 74
5.5 (a) Depth-First Order and the final mapping for the example of Figure 5.4; (b) A zig-zag traversal; (c) A new zig-zag traversal. 75
5.6 (a) Local placement options; (b-c) Two different output traversals (dashed blue line) for the same input traversal from Figure 5.5(b). 76
5.7 Reconvergent example: (a) Depth-first; (b) Zig-zag; (c) Zig-zag plus annotation and optimal mapping. 77
5.8 First Traversal and annotations: (a) Detect a reconvergent or cycle; (b) Create an annotation; (c-d) backpropagate the annotation to the predecessor edges in reversal traversal order. 77
5.9 (a) Data-flow Graph plus I/O annotation for node f; (b) Placement without annotations; (c) Placement guided by I/O annotations. 78
5.10 Degree Matching: (a) Data-flow example; (b) Mesh after the placement of a and b; (c) Number of free cell if c is placed in the dotted boxes; (d) Best option to place c to satisfy the node degree. 79
5.11 (a) Annotated Data-flow. (b) The placement of c without look-ahead; (c) Number of free cells at distance one of a to place c followed by the placement of d; (d) The placement of c guided with look-ahead. 79
5.12 Pseudo-code of first traversal and annotation step. 80
5.13 Pseudo-code of the second traversal and placement step 81
5.14 Edge distribution categories and a placement:(a) A YOTO instance; (b-c) Two YOTT Instances. 85
5.15 Optimal Edges in percentage for (a): SA-100 and YOTT-100; (b) VPR BB and Fast, and and YOTT-100. 87
5.16 Optimal Edges in percentage for YOTT-100 versus (a) YOTO-1000; (b) YOTO-100. 88
5.17 Placement Execution Time: SA-based, YOTO, and YOTT. Low is better. 88
5.18 Maximal FIFO size for VPR BB and Fast, YOTT-100 and SA-100. Low is better. 90
5.19 Maximum FIFO Size for: YOTO-100 and -1000, YOTT-100. Low is better. 90
5.20 Maximal FIFO size with/without Reconvergent Annotation. 91

LISTA DE TABELAS

4.1 Execution Time for a set of benchmarks (Chin et al., 2017a). We compare our traversal proposal with CG_1 (Chin and Anderson, 2018a), CG_2 (Walker and Anderson, 2019a), and CGRA-ME. 64
4.2 Speed-up evaluation of our Fully pipelined P&R. 65
4.3 Six Strategies Worst Case and Geometric Means for the FIFO size in a Mesh and an 1-hop topology 67

5.1 Speedup Ratio (in comparison against VPR), Average FIFO Size, Average Wire Length in Segments 83
5.2 Three Edge Categories: First, Reconvergent an I/Os. 85
5.3 A set of dataflow graph benchmarks provided by UCSB (University of California, 2020) and CGRA-ME framework (Chin et al., 2017a) 86

6.1 Tabela comparativa YOTT versus SA 94

SUMÁRIO

1 Introdução **14**
1.1 Motivação . . . 15
1.2 Justificativa . . . 15
1.3 Objetivo Geral . . . 17
1.3.1 Objetivos específicos . . . 17
1.4 Problema . . . 17
1.4.1 Grafo de fluxo de dados . . . 17
1.4.2 As Três etapas . . . 18
1.5 Solução Proposta . . . 18
1.6 Estrutura da Dissertação . . . 19

2 Background **21**
2.1 Coarse-Grained Reconfigurable Architecture (CGRA) . . . 21
2.1.1 Elemento de Processamento (PE) . . . 23
2.2 Mapeamento em CGRA . . . 24
2.2.1 Escalonamento . . . 28
2.2.2 Técnicas de balanceamento de caminhos . . . 29

3 Algoritmos de Travessia **31**
3.1 Trabalhos Relacionados . . . 31
3.2 Algoritmo baseado em travessia de grafos . . . 37
3.2.1 Algoritmos baseados em profundidade . . . 37
3.2.2 Algoritmo YOTO . . . 38
3.3 Algoritmo YOTT . . . 39

4 TRAVERSAL: A Fast and Adaptive Graph-based Placement and Routing for CGRAs **42**
4.1 Introduction . . . 42
4.2 Background . . . 44
4.2.1 Coarse-Grained Reconfigurable Architecture . . . 44
4.2.2 Latency Balancing during P&R . . . 45
4.2.3 Modulo Scheduling . . . 45
4.2.4 Previous CGRA P&R Research . . . 46
4.3 Traversal Placement . . . 48
4.3.1 Definitions for the Problem . . . 48
4.3.2 P&R Algorithm . . . 50
Input Traversal List P&R . . . 51
Routing of the remaining edges . . . 52
Solution Space Exploration . . . 52
4.3.3 Zigzag Input Traversal . . . 53
4.3.4 Adjacency Lists . . . 55
4.3.5 Interconnection Topologies . . . 56
4.3.6 Modulo Scheduling . . . 56
4.3.7 Elastic CGRA . . . 57

4.4 GPU Graph-based Placement . . . 59
4.5 Experimental Results . . . 61
4.5.1 Target Architectures . . . 61
4.5.2 P&R Algorithms . . . 61
Previous Work . . . 61
Our approaches . . . 62
4.5.3 Execution Time Measurement . . . 63
4.5.4 Metrics . . . 63
4.5.5 Results for a Modulo Scheduling CGRA . . . 63
4.5.6 Results for a Fully Pipelined CGRA . . . 64
4.6 Conclusion . . . 68

5 You Only Look Twice: A YOLT Placement and Routing Approach for CGRAs **69**
5.1 Introduction . . . 69
5.2 Background . . . 71
5.2.1 Mind the gap . . . 71
5.2.2 YOTO: You Only Traversal Once . . . 73
5.2.3 Input Graph Traversal Order . . . 74
5.2.4 Output Graph Traversal Order . . . 75
5.3 YOTT: You Only Traversal Twice . . . 76
5.3.1 First Input Traversal: Learning and Annotating . . . 76
Reconvergent Annotation . . . 77
I/O Annotation . . . 78
5.3.2 Second step: Placement Guidance . . . 78
Degree Matching . . . 78
Look-ahead . . . 79
5.3.3 Algorithm . . . 80
5.4 Experimental Results . . . 82
5.4.1 Algorithm Evaluation Summary . . . 83
5.4.2 Looking under the Hood: Average Wire Length . . . 84
5.4.3 Edge Distribution . . . 86
5.4.4 Percentage of Optimal Edges . . . 87
5.4.5 Execution Time and Number of Swaps . . . 88
5.4.6 Delay Mismatch . . . 89
5.4.7 Reconvergent Annotation . . . 90
5.5 Related Work . . . 90
5.6 Conclusion . . . 92

6 Conclusão **93**

Referências Bibliográficas **96**

Capítulo 1

Introdução

Com um novo perfil de aplicações: redes neurais, aprendizado de máquina, bioinformática e processamento massivo de dados (Zhang and Li, 2017; da Silva et al., 2017; Gao, 2020), a computação atualmente tem como principais desafios prover desempenho e eficiência energética. A desaceleração da lei de Moore em 2015 e o fim da escalabilidade de Dennard por volta de 2005, criou uma demanda por novas soluções na área de arquitetura de computadores (Theis and Wong, 2017; Zhang, 2019). Uma solução eficiente em desempenho é o desenvolvimento de circuitos para aplicações específicas (ASICs) (Boutros et al., 2018), porém esta alternativa tem um alto custo de projeto e não oferece flexibilidade para se adaptar ao surgimento de novas aplicações. Portanto, a eficiência energética e a flexibilidade podem ser consideradas como os critérios principais no desenvolvimento de novas soluções (Liu et al., 2019b).

Processadores de propósito geral (CPUs), unidades de processamento gráficos (GPUs) e processadores de sinais digitais (DSPs) oferecem flexibilidade mas possuem uma baixa eficiência energética. Circuitos programáveis após a fabricação como os *Field Programmable Gate Arrays* (FPGAs) tem eficiência e flexibilidade. Entretanto, desenvolver para FPGA requer conhecimento especializado em hardware, experiência com linguagens de descrição (VHDL, Verilog) e a compilação pode demorar minutos à horas. Por prover conexões e lógica programável após a fabricação, a frequência de clock do FPGA é 10x mais lenta, quando comparada com a dos ASICs (Georgis et al., 2019; HajiRassouliha et al., 2018). Mesmo com o uso das novas técnicas de síntese de alto nível (HLS), a partir de código em C/C++ com anotações, ainda há uma lacuna para implementação de algoritmos em FPGAs, sendo necessário muitas intervenções por parte do desenvolvedor com conhecimentos sobre o processo de síntese e o hardware alvo (Schafer and Wang, 2019).

A partir da década de 90, um novo conceito de computação reconfigurável foi proposto com a introdução da granularidade mais grossa (*Coarse-Grained Reconfigurable Architectures* - CGRA). Diferente do FPGA que é reconfigurado a nível de bit, os CGRAs trabalham no nível de palavras e vem ganhando interesse acadêmico e da indústria devido a sua eficiência energética ser próxima a um ASIC, aliada a flexibilidade próxima a um FPGA. Além disso, o tempo de reconfiguração de um CGRA está

na faixa de $ns - \mu s$ enquanto que um FPGA leva na faixa de $ms - s$ (Liu et al., 2019b).

Nas últimas duas décadas, os CGRAs reduziram o consumo de energia, passando 10W para 0.1W, enquanto o ciclo de *clock* aumentou de 100 MHz para 500 Mhz. Já as GPUs, nesse mesmo período, passaram de 50W para 200W e o *clock* aumentou de 50 MHz para 1500 MHz. Em relação a eficiência energética, os CGRAs tiveram, em média, uma melhora de 0.5 Gops/W para 200 Gops/W enquanto que as GPUs variaram de 0.5 Gops/W para 80 Gops/W (Podobas et al., 2020).

1.1 Motivação

Entretanto ainda faltam ferramentas para a popularização dos CGRAs, dentre elas podemos citar os algoritmos de posicionamento e roteamento (P&R). O problema de P&R para CGRAs é um problema NP-Completo, onde diversos trabalhos propuseram soluções exatas e heurísticas: *Simulated Annealing* (Oliveira et al., 2020; Carvalho et al., 2020), particionamento (Park et al., 2008), técnicas de programação linear (Chin et al., 2017b), algoritmos genéticos (Tukur et al., 2019) e dentre vários outros (Liu et al., 2019b; Kim et al., 2011a; Wang et al., 2017). Contudo, o tempo de execução do mapeamento requer segundos, minutos e até horas para maioria das soluções propostas. Ademais, estes trabalhos são limitados a grafos pequenos (20 a 100 vértices).

Além disso, CGRAs são vistos na academia como um forte candidato a fabricação de computadores de atuais (Prabhakar et al., 2017; Yu et al., 2020) e para aplicações em *datacenters* (Ijaz et al., 2020). Na indústria ainda existem poucos CGRAs, como por exemplo o *WAVE Computing* (Nicol, 2017). Este CGRA é focado em soluções em aplicações para *datacenters*. A Samsung incorporou aceleradores em CGRAs para gerar economia em suas TVs com alta definição de 8K (Kim et al., 2015).

O P&R para CGRA inclui uma dificuldade adicional em comparação com os FPGA que é o balanceamento de caminhos. Portanto, além de posicionar os elementos e interconectá-los (roteamento), deve se ainda equilibrar o comprimento dos caminhos no pipeline para sincronizar corretamente os fluxos de dados.

1.2 Justificativa

Para arquiteturas com múltiplos UPs, o mapeamento estático é mais vantajoso pois pode produzir a computação de todo o grafo a cada ciclo. Além do número de UPs, outros fatores irão influenciar o custo da arquitetura. Os principais fatores são os recursos de roteamento entre as UPs e os recursos de temporização. Neste contexto, Nowatzki et al. (2018a) analisa a relação entre o tempo de P&R com escalonamento e a qualidade da solução de vários algoritmos de mapeamento em dois cenários de hardware. Os resultados estão ilustrados na Figura 1.1. A qualidade é avaliada em função

da vazão, onde o valor 1 corresponde a vazão máxima, produzindo uma solução a cada ciclo de *clock*. Uma vazão 2 significa um $II = 2$, onde tem-se uma nova computação a cada 2 ciclos, ou seja, uma redução da vazão por um fator 2. Adicionou-se a abordagem YOTT proposta nesta dissertação para comparar com as três abordagens avaliadas em (Nowatzki et al., 2018a) que são: (1) heurística gulosa; (2) solução exata; (3) heurística híbrida. A Figura 1.1(a) considera uma arquitetura alvo com um custo extra 50% em área para disponibilizar filas (FIFOs) elásticas de até 15 entradas nos elementos de processamento. Estas FIFOs são usadas para equilibrar os caminhos de dados simplificando o algoritmo de P&R. Neste cenário a heurística gulosa é bem rápida (da ordem de milissegundos), mas não encontra solução para 33% dos *benchmarks* avaliados. A estratégia exata não encontra solução para 40% dos casos avaliados e em média executa em 100 segundos. A solução híbrida requer de 10-100 segundos e encontra solução para todos os casos avaliados. A solução proposta nesta dissertação é quase tão rápida quanto a solução gulosa e consegue resolver com sucesso todos os casos.

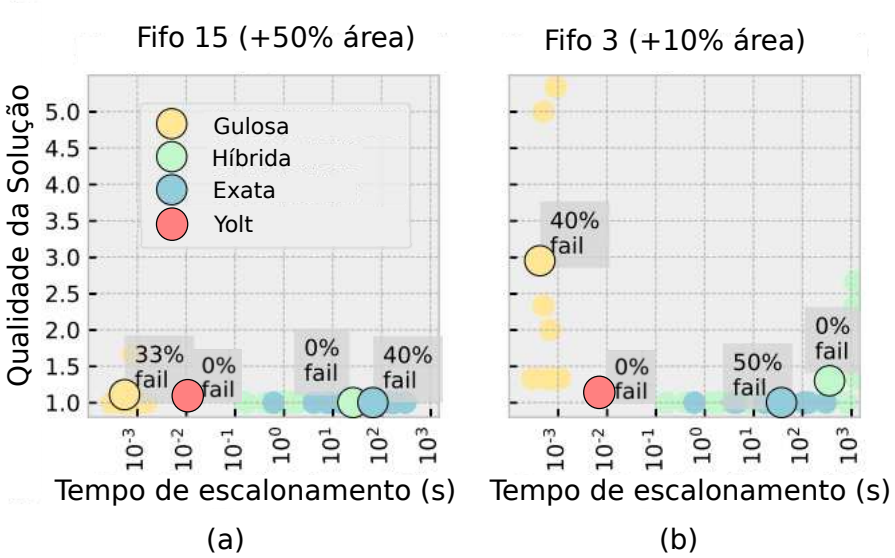

Figura 1.1: Qualidade da solução em função do tempo de escalonamento: (a) Cenário com uma Arquitetura com custo adicional de 50% e FIFOS elásticas de tamanho 15; (b) Cenário com uma arquitetura com FIFO elásticas de Tamanho 3 e custo adicional de 10%. Figura extraída e adaptada de Nowatzki et al. (2018a)

A Figura 1.1(b) avalia um segundo cenário. A área é incrementada em apenas 10% para incluir FIFOs com o comprimento máximo de 3 elementos. A vazão da solução gulosa fica três vezes pior e ainda falha para 40% dos casos avaliados. A solução exata encontra a vazão ótima mas falha para 50% dos casos e o tempo de execução é da ordem de 100 segundos. A solução híbrida encontra a solução com uma pequena redução na vazão para todos os casos avaliados, porém o tempo de execução é da ordem de 1000 segundos.

Uma questão que pode ser levantada é se existe a possibilidade de reduzir o tempo de execução no cenário da Figura 1.1(b) com FIFOs de até 3 elementos e ainda assim encontrar uma solução. Outro ponto importante que não foi destacado na Figura 1.1 é o tamanho dos grafos avaliados. Os *benchmarks* tem de 20-30 nodos. Como já mencionado, a exigência de balanceamento reduz muito o número de soluções válidas no espaço de 20! à 30! possibilidades, ou seja, mesmo para grafos com poucos vértices o espaço é muito grande variando de 10^{18} a 10^{32} possibilidades. Nesta dissertação apresentamos soluções que tratam grafos maiores em um tempo de execução de milissegundos com a restrição de FIFOs pequenas.

1.3 Objetivo Geral

O objetivo dessa dissertação é o desenvolvimento de algoritmos de P&R para CGRAs com tempo de execução na ordem de milissegundos.

1.3.1 Objetivos específicos

- Desenvolver novas heurísticas para o problema de P&R.
- Comparar com as ferramentas do estado da arte.
- Explorar as propriedades dos grafos para aumentar a eficiência do P&R.
- Avaliar CGRAs com configuração estática e dinâmica.

1.4 Problema

Esta seção detalha o problema de mapeamento para grafos em CGRAs, que foi introduzido na Seção 2.2. O mapeamento envolve três etapas: posicionamento, roteamento e escalonamento. A capacidade de posicionar, rotear e escalonar depende tanto da estrutura física da arquitetura quando do algoritmo de mapeamento e a integração entre as suas três etapas.

1.4.1 Grafo de fluxo de dados

A Figura 1.2(a) apresenta um exemplo de um grafo de instruções. Onde os nodos azuis são do tipo entrada, o nodo vermelho é do tipo saída e os nodos intermediários são os nodos de instruções. A Figura 1.2(b) exemplifica um grafo posicionado e roteado em um CGRA 3×3 utilizando o mapeamento estático.

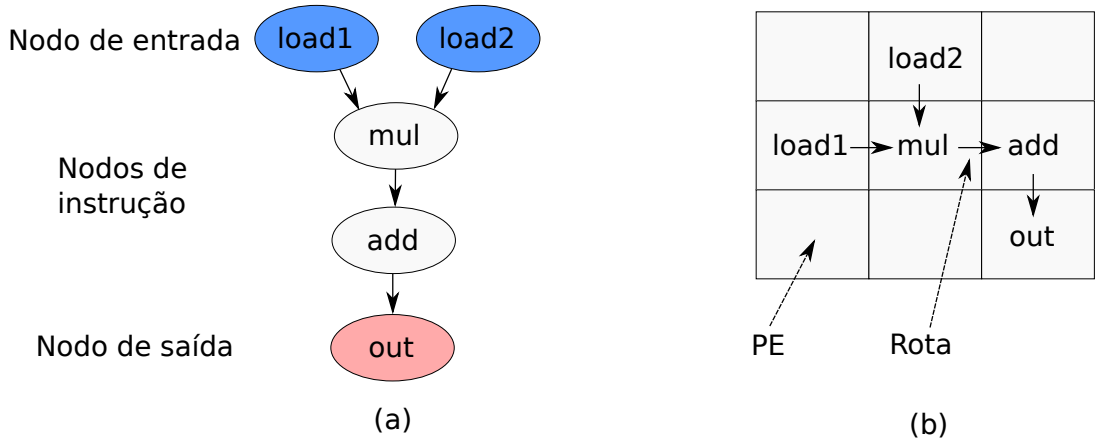

Figura 1.2: (a) Grafo de fluxo de dados; (b) P&R do grafo.

1.4.2 As Três etapas

Considerando o mapeamento estático. Como mencionado, o mapeamento envolve três etapas:

- **Posicionamento**: Atribuir uma unidade de processamento para cada nodo do grafo de fluxo de dados;
- **Roteamento**: Atribuir recursos de interconexão para prover a comunicação entre os nodos do grafo programando os recursos de roteamento;
- **Escalonamento**: Ajustar a temporização do fluxo dos dados para garantir a correta execução do grafo inicial.

1.5 Solução Proposta

Existem diversas soluções na literatura para P&R. Técnicas utilizando ILP (Chin et al., 2017b; Chin and Anderson, 2018a; Walker and Anderson, 2019b) e SAT Solver (Park et al., 2009b; Nadel, 2017) são limitadas a grafos com 20 nodos ou menos. Técnicas híbridas também não escalam para grafos maiores (Nowatzki et al., 2018a). Soluções utilizando SA resolvem com um custo adicional no tempo de compilação (Murray et al., 2020; Oliveira et al., 2020). Algoritmos baseados na travessia de grafos (Ferreira et al., 2015; Chhugani et al., 2012) podem resolver rápido, mas precisam de mais recursos no hardware, com FIFOs para rebalancear o grafo. Portanto, esse trabalho foca em melhorar os algoritmos de travessia reduzindo os recursos de hardware.

Para o problema P&R para CGRAs foram desenvolvidos duas propostas: Traversal (Seção 3.2) e o YOTT (Seção 3.3). Ambos utilizam algoritmos de travessia de grafos, onde tanto o grafo de fluxo de dados como o grafo que representa a arquitetura alvo são visitados. A abordagem Traversal propõe um algoritmo que se baseia na técnica

YOTO (***Y**ou **o**nly **l**ook **o**nce*), onde a travessia é executada uma única vez. Já a proposta YOTT recebe esta denominação, ***Y**ou **o**nly **l**ook **t**wice*, por apresentar uma abordagem que executa a travessia duas vezes.

1.6 Estrutura da Dissertação

Esta dissertação está estruturada conforme o formato de coletânea de artigos científicos normalizado pelo Conselho Técnico de Pós-Graduação da Universidade Federal de Viçosa. Este formato é composto pela introdução geral do problema, dos capítulos provindos de artigos e de uma conclusão geral. Especificamente, capítulo 1 apresenta uma introdução geral que abrange os temas dos artigos que compõe a dissertação e uma motivação e justificativa o problema P&R com CGRA.

Capítulo 2 apresenta o referencial teórico sobre as arquiteturas de CGRA, a estrutura interna dos elementos de processamento e as estruturas de interconexão. Além disso, a seção 2.1 apresenta o CGRA, formalizando o tipo de arquitetura com seus principais componentes e comparando-o com outras arquiteturas no requisito de eficiência energética e flexibilidade. A seção 2.2 descreve o mapeamento estático e dinâmico de uma aplicação em um CGRA. O mapeamento dinâmico utiliza o modelo MRRG (*Modulo Routing Resource Graph*), no qual é possível mostrar que independente do intervalo de inicialização, o problema a ser resolvido é um mapeamento de grafos. Finalmente, a seção 2.2.1 apresenta o escalonamento no mapeamento estático e a seção 2.2.2 ilustra três técnicas de balanceamentos dos caminhos para resolver os problemas de sincronização.

Capítulo 3 faz uma discussão dos algoritmos de posicionamento e roteamento (P&R) para grafos de fluxo de dados. A seção 3.1 introduz os principais trabalhos que introduziram os algoritmos de travessias de grafos para resolver o problema de P&R. A seção 3.2 apresenta o princípio de funcionamento de um algoritmo baseado em travessia de grafos e as tomadas de decisões necessárias durante o mapeamento no grafo de arquitetura. A seção 3.2.1 exemplifica algoritmos baseados na travessia em profundidade. A seção 3.2.2 apresenta a classe de algoritmos YOTO que utilizam uma abordagem gulosa em seu mapeamento. Enfim, a seção 3.3 introduz o algoritmo YOTT, proposta nesta dissertação, juntamente com sua estratégia de percusso duplo sobre o grafo da aplicação, onde o primeiro percurso gera uma lista de anotações que irá guiar o segundo percurso, auxiliando nas escolhas e solucionando parcialmente o problema de reconvergência dos grafos.

Capitulo 4 apresenta um novo algoritmo de P&R baseado em travessia por grafo. Os resultados mostraram que a nova abordagem com o uso do modulo *scheduling* gera mapeamentos ótimos tendo uma melhora cinco ordens de execução comparado com o *framework* CGRA-ME (Chin et al., 2017a). Além disso, considerando o mapeamento

espacial na escala de milissegundos, a implementação com GPU do algoritmo de P&R é 10x em média mais rápida em comparação com a ferramenta VPR (Murray et al., 2020). As contribuições desse artigo foram:

- Manipulação de Múltiplas saídas em um grafo.
- Busca sistemática no espaço de solução;
- Exploração com GPUs.

Capitulo 5 introduz um novo algoritmo de P&R denominado YOTT que explora melhor o espaço de busca. Diferente da abordagem YOTO apresentada no capítulo 4 em que as decisões são aleatórias e gulosas, YOTT explora as propriedades do grafo para evita grandes buscas aleatórias. Os resultados mostraram que a abordagem YOTT é 83.4x e 19.4x mais rápida do que VPR e YOTO, respectivamente. Além disso, o YOTT reduz o comprimento médio do fio e o tamanho máximo das FIFOs. As contribuições desse artigo foram:

- Anotações de listas de travessias.
- Método sistemático para transferir a localidade do grafo de entrada para saída;
- Melhor qualidade nos posicionamentos e roteamentos comparado com as abordagens YOTO e VPR.

Por fim, no capítulo 6 é apresentado uma conclusão geral e trabalhos futuros.

Capítulo 2

Background

CGRA é uma classe de arquitetura reconfigurável a nível de palavra, oferecendo flexibilidade e desempenho próximo a uma computação de propósito geral (Zhou et al., 2013a). A arquitetura do CGRA é composta por um conjunto de elementos de processamento e uma topologia de interconexão. Existem diversas arquiteturas (Ferreira et al., 2018): malha, cubo, lineares, em níveis, etc. As tridimensionais baseadas em cubo são poucos usuais (Karunaratne et al., 2017). Outras arquiteturas (Zhou et al., 2013b) tem hierarquia com o uso de *clusters* em sua estrutura. Nas arquiteturas bidimensionais no plano, as arquiteturas mais populares são em grade ou 1-hop, mas existem também trabalhos que exploram as conexões em colmeia com conexões hexagonais (Silva et al., 2006). De acordo com Hartenstein (2001), as três principais arquiteturas são: (1) Uni-dimensional linear com barramento ou em pipeline; (2) Em grade com conexões 1-hop ou outros padrões; (3) Em malha simples que é a mais comum dentre todas.

Além da topologia, outro ponto importante é o modelo de execução do CGRA: estático ou dinâmico. No modelo estático, cada elemento de processamento recebe uma operação em tempo de compilação e irá executar a mesma operação até o término da execução. No modelo dinâmico, cada elemento recebe um conjunto de palavras de configuração e realiza uma multiplexação no tempo durante a execução, podendo executar uma operação diferente a cada ciclo.

2.1 Coarse-Grained Reconfigurable Architecture (CGRA)

CGRAs podem alcançar eficiência energética similar aos arranjos sistólicos que são arquiteturas fixas e específicas, definidas no tempo de projeto. O CGRA possui menos flexibilidade que o FPGA, pois os FPGAs trabalham a nível de bit, enquanto que um CGRA trabalha a nível de palavra. Porém o CGRA é mais simples para ser reconfigurado e mapeado. Já os circuitos integrados para uma aplicação específica (ASICs) oferecem uma alta eficiência energética, mas não oferecem flexibilidade, similar aos sistólicos. Finalmente, os processadores de propósito geral (CPUs), unidades de processamento gráfico (GPUs) e processadores de sinais digitais (DSPs) apresentam fle-

xibilidade a nível de software, mas possuem baixa eficiência energética. Portanto, os CGRAs se posicionam como uma arquitetura com flexibilidade e eficiência energética complementar as outras arquiteturas.

CGRAs são uma classe de processadores reconfiguráveis para computação a nível de palavra ou instrução (Liu et al., 2019b). O aumento da granularidade reduz a complexidade e elimina algumas limitações dos FPGAs em comparação com a computação de propósito geral (Podobas et al., 2020). Ao trabalhar com largura de bits de tamanhos distintos (8, 16, 32 bits) e encapsular operações a nível de palavra, o CGRA apresenta vantagens sobre FPGA que trabalha a nível de bit. A definição de uma arquitetura CGRA tem três etapas. A primeira etapa é a modelagem da arquitetura, criando uma matriz de elementos de processamento (PE) interconectados por uma rede, juntamente com interconexões com entradas, saídas e blocos de memória. A segunda etapa define a granularidade e as operações dos PEs. A última etapa cria uma instância do CGRA em função dos parâmetros mencionados, permitindo que o CGRA seja sintetizado e/ou simulado (Tehre and Kshirsagar, 2012).

A Figura 2.1 mostra um CGRA em malha ou grade de tamanho de 4×4 PEs. Cada PE pode executar uma operação aritmética ou lógica, como adição, deslocamento, multiplicação, carregar, armazenar e dentre outras. Os PEs podem trocar dados com a memória local do processador. A cada ciclo, o PE pode interagir com o PE vizinho enviando ou recebendo dados.

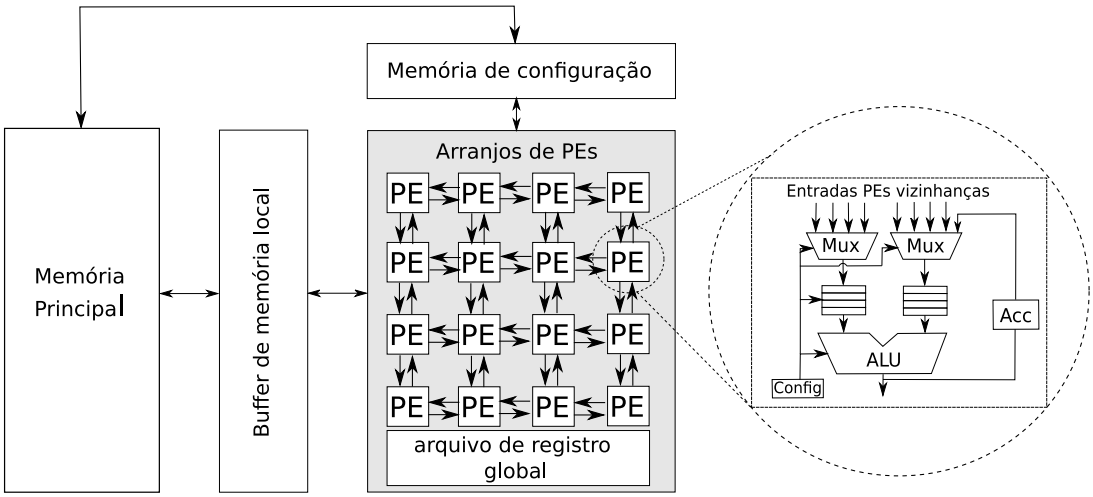

Figura 2.1: Estrutura básica de um CGRA bi-dimensional conectado entre os seus vizinhos de tamanho 4×4. Adaptado de (Kim et al., 2011b).

A Figura 2.2 ilustra exemplos de topologias de interconexões. A Figura 2.2(a) apresenta uma topologia uni-dimensional linear com barramento ou em pipeline. Esta topologia é utilizada em aplicação de multimídia (Lee et al., 2008) ou em aplicações de processamento de sinal (Eguro, 2002). A Figura 2.2(b) mostra a topologia malha que é a mais usada em CGRAs (Vieira et al., 2021; Chin et al., 2017b). A Figura 2.2(c) ilustra

a interconexão 1-hop (Oliveira et al., 2020), uma variação da topologia malha, onde cada PE liga em oito vizinhos, dois em cada direção e que foi utilizada nesse trabalho. Outras topologias também são possíveis de serem construídas. A Figura 2.3(a) mostra uma topologia em cluster (Baumgarte et al., 2003). A Figura 2.3(b) apresenta a topologia strip (Ferreira et al., 2011a) onde cada camada de PE comunica com a próxima camada. A Figura 2.3(c) ilustra uma topologia que utiliza uma rede de interconexão de multiestágio (Silva et al., 2019). Finalmente, a Figura 2.3(d) exemplifica uma topologia tridimensional em formato de cubo (Karunaratne et al., 2017). Portanto existem diversas possibilidades, onde podemos destacar a topologia em grade com conexões 1-hop (Bansal et al., 2004a).

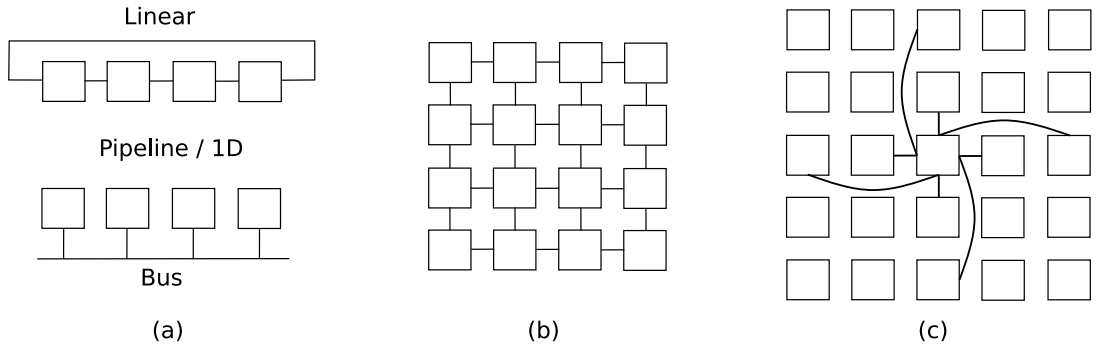

Figura 2.2: Topologias e interconexões: (a) linear; (b) malha; (c) 1-hop;

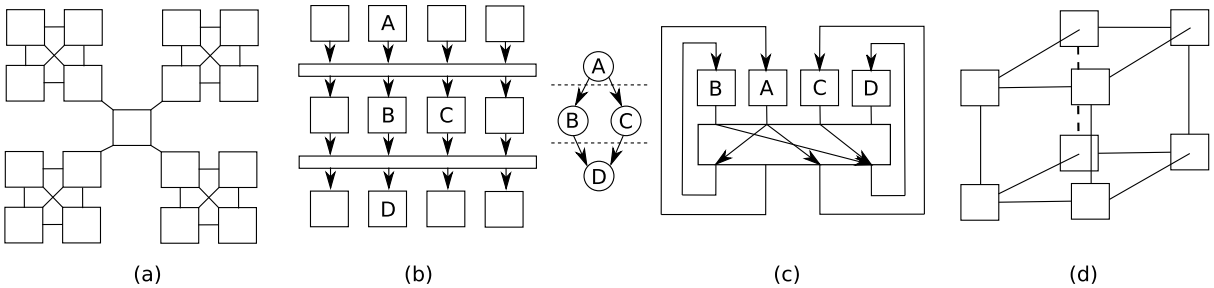

Figura 2.3: Topologias e interconexões mais complexas: (a) cluster; (b) strip; (c) multistágio; (d) cubo.

2.1.1 Elemento de Processamento (PE)

Um CGRA é formado por um conjunto de unidades de processamento ou UPs. Cada UP executa, de forma estática ou dinâmica, um conjunto de operações lógicas, aritméticas e/ou memória. Cada UP está inserido dentro de um elemento de processamento (PE). O PE inclui a UP, os recursos de roteamento (multiplexadores e fios) para conectar aos vizinhos e recursos de balanceamento de caminhos (registradores e/ou FIFOs).

Um PE estático é configurado para executar uma função na sua UP assim como os recursos de roteamento são programados para enviar e receber os dados aos PEs vizinhos, enquanto que um PE dinâmico permite a reconfiguração durante a execução. Desta maneira múltiplas instruções utilizarão o mesmo recurso que será multiplexado no tempo, usando por exemplo a técnica de modulo *scheduling*.

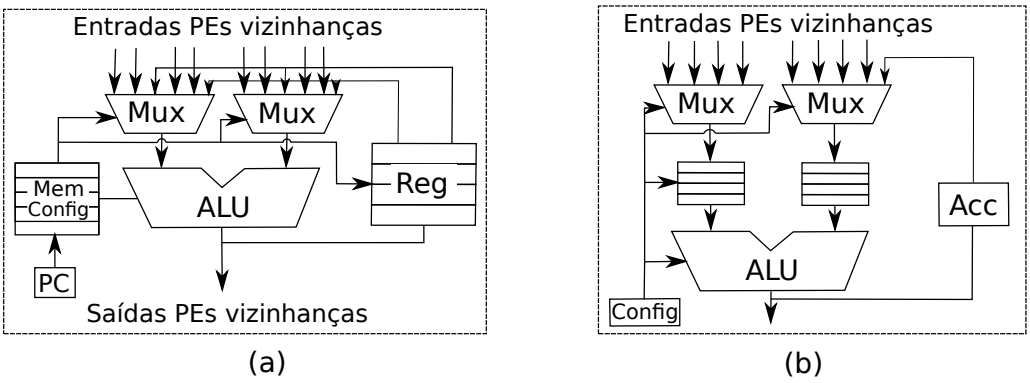

Figura 2.4: Arquiteturas PE: (a) dinâmico; (b) estático.

A Figura 2.4 apresenta dois exemplos de PEs, sendo um PE estático e outro PE dinâmico. Neste exemplo, os dois tipos tem os mesmos recursos de roteamento implementados pelos multiplexadores. Cada multiplexador pode receber dados dos PEs vizinhos. No PE estático, o multiplexador pode receber o dado de um registro interno e no PE dinâmico de um banco de registradores. A unidade de processamento (UP) é implementada com uma unidade lógica aritmética (ALU). O PE dinâmico, apresentado na Figura 2.4(a), possui um contador de programa (PC) para selecionar em tempo de execução a operação da ALU e a configuração de roteamento. Além disso, também é necessário configurar o banco de registradores. Já o PE estático, ilustrado na Figura 2.4(b), possui apenas uma palavra de configuração que permanecerá a mesma durante a execução. Outra vantagem do PE estático é eliminar a busca e decodificação das instruções a cada ciclo, resultando em uma economia de energia. Além disso, o PE estático da Figura 2.4(b) possui FIFOs na entrada da ALU. Este recurso pode ser usado para o escalonamento, que será detalhado na próxima seção. Neste exemplo, o PE dinâmico pode fazer uso dos registradores para realizar o escalonamento, se necessário.

2.2 Mapeamento em CGRA

Uma aplicação descrita como um grafo de fluxo de dados pode ser mapeada em um CGRA de forma estática ou dinâmica. No mapeamento estático, o compilador irá gerar uma palavra de configuração para cada unidade de processamento (UP) do CGRA. Uma vez configurada, a UP irá permanecer com esta função até o término da

execução. Por outro lado, o compilador irá escalonar mais de uma palavra de configuração por UP no mapeamento dinâmico. Durante a execução, a cada ciclo, a UP pode ser reconfigurada para executar outra funcionalidade, assim como as conexões entre as UPs.

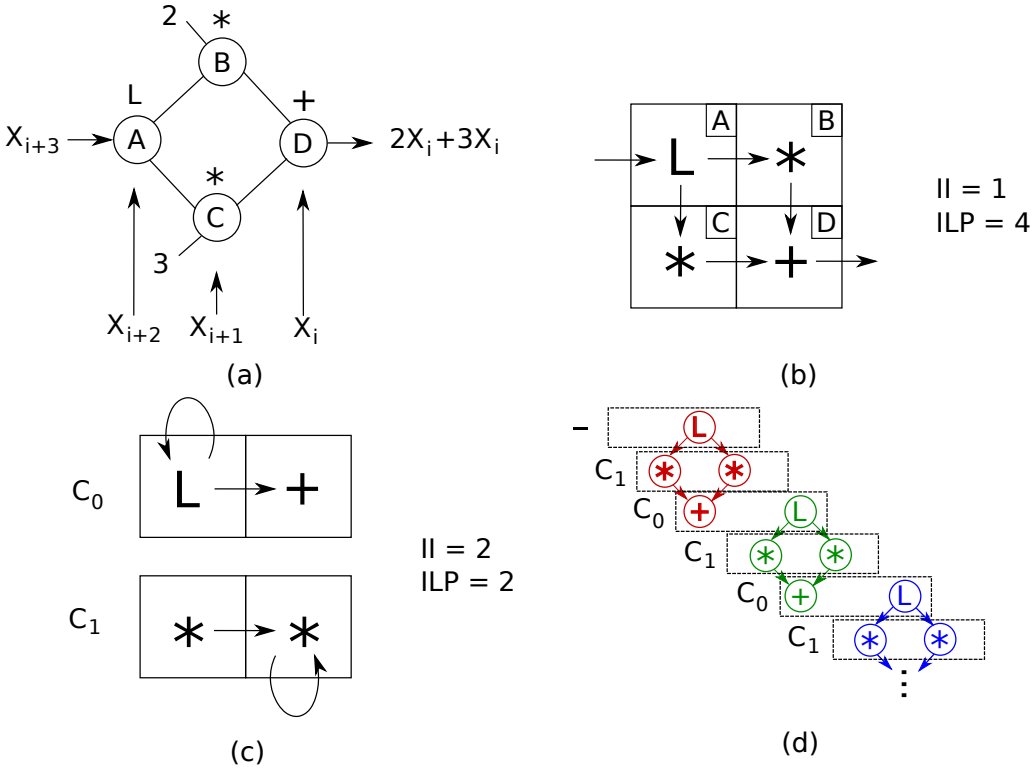

Figura 2.5: Mapeamento em um CGRA: (a) grafo; (b) mapeamento estático; (c) mapeamento dinâmico; (d) Modulo Scheduling.

A Figura 2.5 apresenta um exemplo de mapeamento para CGRA para computação da expressão $2X_i + 3X_i$, onde X é um vetor cujo os dados são inseridos no grafo. A cada ciclo, um novo elemento i do vetor X é aplicado a entrada do grafo. Suponha que cada operação tenha a latência de um ciclo. A Figura 2.5(a) mostra que enquanto o elemento X_{i+3} é preparado para entrar no grafo, o nodo A processa o elemento X_{i+2}, os nodos B e C, em paralelo, estão processando os valores gerados pelo elemento X_{i+1} e finalmente, o nodo D está processando o elemento X_i. Pode-se observar o paralelismo espacial e temporal da execução em pipeline.

Os nodos A, B, C e D são responsáveis em computar as operações de leitura (*load*), multiplicação por 2, multiplicação por 3 e adição, respectivamente. A Figura 2.5(b) mostra uma solução de mapeamento estático para uma arquitetura em malha. A cada ciclo, um novo elemento é inserido, portanto o intervalo de inicialização (II) é igual a 1. A latência é de 3 ciclos, no qual após o primeiro resultado, a pipeline está cheia e produz uma nova computação por ciclo. Portanto, a vazão é de uma expressão computada por ciclo. Outra medida é o paralelismo a nível de instrução (*instruction*

level paralelism - ILP). O hardware possui 4 unidades e todas estão ativas a cada ciclo, gerando um ILP=4.

Pode ocorrer que o grafo tenha mais operações que a arquitetura alvo. Neste cenário, as UPs da arquitetura podem ser multiplexadas no tempo para executar a computação do grafo. A Figura 2.5(c) apresenta o mapeamento para um CGRA com apenas duas UPs para a computação da expressão $2X_i + 3X_i$. Agora serão necessários pelo menos dois ciclos para produzir uma nova computação da expressão completa. O II é igual a 2, ou seja, a cada 2 ciclos, um novo elemento X_i é inserido. O ILP é de 2 operações por ciclo e a ocupação das UPs é máxima. Este exemplo utiliza a técnica de modulo scheduling, que é um método de otimização que expande o laço de computação e sobrepõe iterações sucessivas no tempo. A Figura 2.9(d) exemplifica três iterações com cores diferentes. Nos ciclos pares, as duas UPs estão configuradas para executar o Load e a Adição, que corresponde ao ciclo C_0 na Figura 2.9(c). Portanto irá executar a última operação da iteração em cor vermelha, a adição, na UP_1 e em paralelo a primeira operação da iteração verde, o Load, na UP_0. No ciclo ímpar, as UPs são dinamicamente reconfiguradas para executarem as multiplicações de uma mesma iteração, denotado como ciclo C_1 na Figura 2.9(c). Este processo se repete, onde após C_1, a execução é reconfigurada para retornar a configuração C_0 para terminar a iteração corrente e começar a próxima.

O mapeamento dinâmico é equivalente a um mapeamento estático se representarmos as conexões temporais utilizando o modelo MRRG (*Modulo Routing Resource Graph*) (Mei et al., 2007). A ideia básica é gerar uma cópia do grafo da arquitetura para cada configuração temporal. As conexões entre os UPs que são espacialmente vizinhos no grafo original da arquitetura serão realizadas entre as partições temporais. Ou seja, se existe a conexão $UP_a \rightarrow UP_b$ no grafo original, teremos a conexão $UP_a(t) \rightarrow UP_b(t+1)$, onde t e $t+1$ são duas partições temporais consecutivas. Para simplificar a notação, iremos usar as variáveis X, Y e Z para partições temporais consecutivas, onde X_i equivale ao $UP_i(t)$ e Y_i ao $UP_i(t+1)$. Iremos considerar que todo UP pode manter o valor, ou seja, $UP_i(t) \rightarrow UP_i(t+1)$ ou $X_i \rightarrow Y_i$.

A Figura 2.6(a) apresenta o mapeamento dinâmico em um grid 2×2 com II=2, sendo X_1 um nodo computacional que faz ligação temporal com ele mesmo Y_1 e seus vizinhos Y_2 e Y_3. A Figura 2.6(b) mostra o grafo no formato MRRG para o grid 2x2 considerando II=2. Pode-se observar as duas partições temporais X e Y que foram representadas com as cores branco e preto. Observe que cada nodo X conecta a três nodos Y e vice-versa. Como mencionado, o MRRG reduz o problema de mapeamento dinâmico a um problema de mapeamento estático no MRRG. Se o grafo original tem n nodos, o MRRG terá $II \times n$ nodos. No nosso exemplo, para $II = 2$, o grid gerou um MRRG com $2 \times 4 = 8$ nodos.

A Figura 2.7 exemplifica um mapeamento usando MRRG. Primeiro, os nodos na

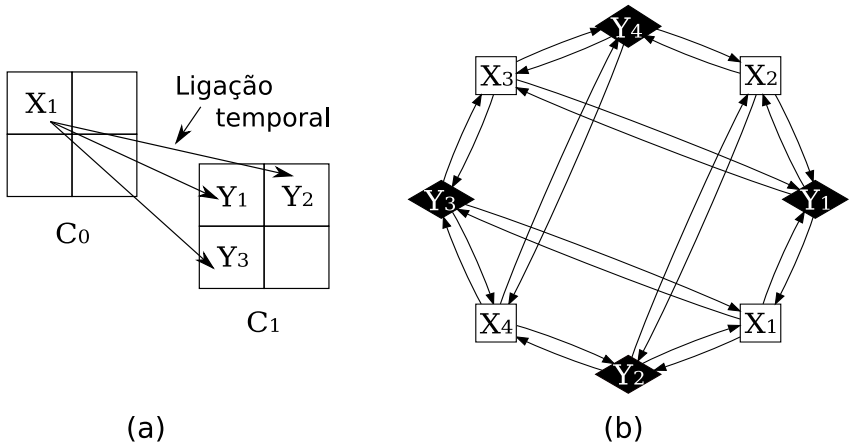

Figura 2.6: Grafo MRRG para uma arquitetura em malha: (a) arquitetura 2×2 com duas partições temporais; (b) MRRG com II=2.

cor branca serão escalonados no tempo C_0 e os nodos em preto serão escalonados no tempo C_1 em um mapeamento dinâmico. A Figura 2.7(b) mostra uma das soluções para o mapeamento com a modelagem baseada em MRRG. O grafo de entrada tem 6 operações, portanto o mapeamento com 2 partições é ótimo, pois é o mínimo para cobrir todas as operações do grafo de entrada. Os nodos do tipo quadrado na cor branca representam a configuração C_0 e os nodos do tipo losango na cor preta a configuração C_1.

Suponha agora uma arquitetura alvo capaz de executar o grafo de entrada da Figura 2.7(a) de forma estática com $II = 1$. No mínimo precisa-se de uma arquitetura com 6 *PEs*. Suponha uma arquitetura $n \times n$. Para mapear 6 operadores, usa-se então uma arquitetura 3×3. A Figura 2.7(c) mostra a arquitetura alvo. Observe que não há partições temporais e o grafo será mapeado de forma estática. A Figura 2.7(d) apresenta o resultado do mapeamento. Resumindo, tanto o mapeamento estático quanto o mapeamento dinâmico com MRRG podem ser modelados como a transformação de um grafo de entrada em um grafo de uma arquitetura alvo.

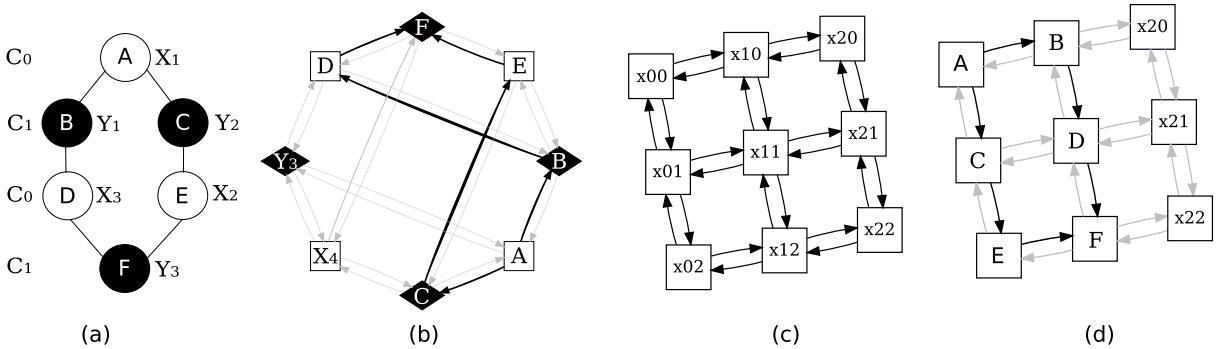

Figura 2.7: Mapeamento dinâmico: (a) Grafo de exemplo de entrada; (b) Mapeado II=2 no MRRG; (c) Arquitetura em malha 3×3; (d) Mapeamento estático.

Na modelagem com MRRG nem sempre é possível mapear um grafo de entrada

com m operações em um MRRG com valor mínimo de II, ou seja, $II = \frac{m}{n}$, onde n é o número de *UPs* da arquitetura. Suponha o grid 2×2 do exemplo anterior. Com o II=2 poderíamos na teoria mapear grafos com até 8 operações. Entretanto, temos que considerar também a partição dos nodos entre as configurações. A Figura 2.8(a) apresenta outro exemplo de grafo. Apesar de ter 8 nodos, a partição na cor branca tem 3 nodos e a partição na cor preta tem 5. Portanto não existe solução com $II = 2$. Neste caso tem-se que aumentar o valor de II para 3, em que resulta em três configurações ilustradas na Figura 2.8(b) com as cores: branco, cinza e preto. A Figura 2.8(c) mostra o mapeamento no MRRG com II=3. Já no mapeamento estático em uma arquitetura com n UPs, onde $n \geq m$, se o P&R for bem sucedido, pode-se mapear o grafo com a vazão máxima como ilustrado na Figura 2.8(d).

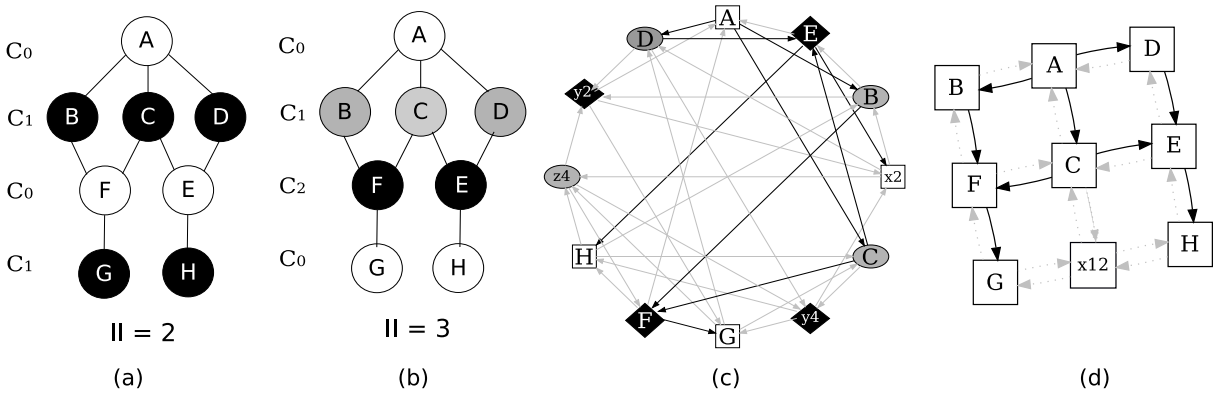

Figura 2.8: Mapeamento dinâmico: (a) Grafo de entrada utilizando II=2; (b) Grafo de entrada utilizando II=3; (c) MRRG com II=3; (d) Mapeamento estático em uma arquitetura 3×3 em malha.

2.2.1 Escalonamento

A etapa de escalonamento tem que garantir a correta sincronização dos dados que estão fluindo em pipeline. Além disso, o escalonamento pode maximizar a vazão. Em um PE dinâmico, a cada ciclo, o escalonamento irá atribuir uma solução de P&R para os nodos que irão ser mapeados naquele instante. Podemos pensar que o grafo será dividido em partições temporais. Cada partição irá mapear uma parte do grafo. Um dos objetivos é minimizar o número de partições que define o intervalo de inicialização (II), que corresponde também a vazão de saída dos resultados.

Para o PE estático, o escalonamento pode ser necessário para implementar o balanceamento dos caminhos para garantir o funcionamento correto. O balanceamento pode ser implementado com filas de tamanho ajustável, PE sendo usado para inserir atrasos ou uso de fios longos para gerar atrasos para equilibrar os caminhos.

A Figura 2.9(a) apresenta um exemplo de grafo de fluxo de dados para a computação da convolução $3X_i + X_{i+1}$. Observe que o nodo C implementa uma operação

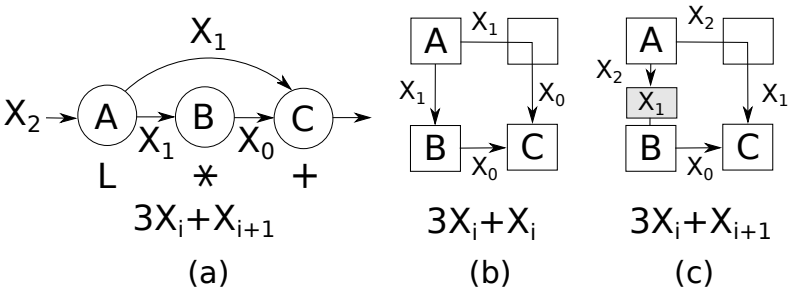

Figura 2.9: Exemplo de Mapeamento: (a) Grafo de entrada; (b) Solução inválida; (c) Solução válida com o uso de uma FIFO.

que trabalha com dois elementos consecutivos X_i e X_{i+1}. Ao realizar o mapeamento é necessário verificar a temporização.

O mapeamento da Figura 2.9(b) gera uma solução de P&R inválida. O caminho $A \rightarrow C$ tem uma latência de 2 ciclos para ser roteado, desequilibrando a computação. A Figura 2.9(c) exemplifica uma solução para esse problema, adicionando uma fila entre os PEs A e B, atrasando em ciclo este caminho para gerar a computação correta em C com os elementos X_i e X_{i+1}. A próxima seção apresenta outras técnicas que podem ser usadas para equilibrar os caminhos.

2.2.2 Técnicas de balanceamento de caminhos

O mapeamento tem que garantir que cada aresta do grafo seja mapeada com latência de um ciclo. Caso alguma aresta seja mapeada utilizando vários segmentos de fios com uma latência maior que uma unidade, todos caminhos devem ser verificados para garantir o sincronismo do fluxo original.

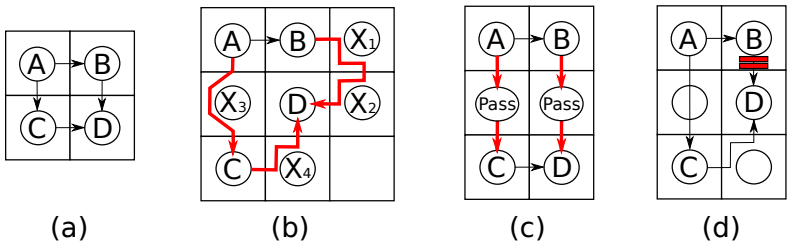

Figura 2.10: Técnicas de balanceamento: (a) Mapeamento ótimo; (b) Caminhos longos; (c) Uso da UP; (d) Filas ajustáveis.

A Figura 2.10 apresenta técnicas para resolver o desbalanceamento. Enquanto a Figura 2.10(a) apresenta um mapeamento ótimo, as Figuras 2.10(b-d) ilustram as três soluções para equilibrar os caminhos quando o P&R não é ótimo. Um exemplo da técnica com caminhos longos é ilustrado na Figura 2.10(b). Suponha que a aresta $A \rightarrow C$ utiliza dois segmentos de fios devido ao posicionamento de A e C em PEs que não são vizinhos. O desbalanceamento irá ocorrer em caminhos reconvergentes. Caso

um dos caminhos tenha uma ou mais arestas implementadas com vários segmentos de fios (passando por vários PEs), os outros caminhos podem ser equilibrados inserindo rotas maiores para equilibrar a propagação do fluxo de dados. Neste exemplo, a aresta $B \rightarrow D$ também usou três segmentos de fios, equilibrando o caminho, devido as arestas $A \rightarrow C$ e $C \rightarrow D$. A latência aumentou de 3 para 5 ciclos, mas a vazão continua de uma computação por ciclo. Neste exemplo podemos observar também que os nodos X_1, X_2, X_3 e X_4 estão realizando outra computação e além disso implementam o roteamento das arestas $A \rightarrow C, C \rightarrow D$ e $B \rightarrow D$.

Quando o recurso de roteamento não está disponível, a UP interna do PE pode ser usada para repassar a computação, como ilustrado na Figura 2.10(c), sem processa-la. A desvantagem desse método é que você desperdiça uma ALU que poderia realizar uma computação. A terceira técnica mencionada utiliza filas ajustáveis dentro do PE para atrasar um ou mais caminhos, como ilustrado na Figura 2.10(d). Neste caso, as arestas $A \rightarrow C$ e $C \rightarrow D$ usam dois segmentos de fios. Como B e D são vizinhos, a solução foi inserir uma fila ajustável com dois *slots* para balancear o outro caminho.

Todas as três técnicas podem ser usadas para garantir o correto funcionamento. Porém a melhor solução é gerar um P&R que minimize o desbalanceamento, uma vez que as filas, uso de fios para atraso ou UP apenas para repassar dados subutilizam e aumentam o custo da arquitetura (Silva et al., 2020).

Capítulo 3

Algoritmos de Travessia

Apesar dos algoritmos de travessia para posicionamento e roteamento (P&R) não serem novidades na literatura (Ferreira et al., 2007), essa dissertação introduz contribuições na área. Primeiro, o algoritmo YOTO apresenta uma nova abordagem para manipulação de grafos com múltiplas saídas. Cada instância de execução do algoritmo pode começar por nodo distinto, resultando em uma solução diferente. Para reduzir o tempo de execução no processo de exploração do espaço de soluções, uma versão utilizando GPUs foi implementada. Além disso, uma busca sistemática no espaço de solução foi criada com o objetivo de melhorar a exploração. Foram utilizados algoritmos de busca profundidade e largura. O YOTO também introduziu um novo percurso de travessia, o ZigZag. Diferente dos algoritmos tradicionais, o ZigZag percorre as arestas nas duas direções com o objetivo de capturar as correlações das múltiplas saídas.

Contudo, o YOTO é um algoritmo que se baseia em decisões aleatórias e gulosas, não resolvendo o problema da reconvergência dos grafos. Deste modo, um novo algoritmo foi desenvolvido, denominado pelo nome de YOTT, com o propósito de explorar as propriedades dos grafos. O YOTT cria anotações na lista de travessia para tratar os caminhos com reconvergência e evitar decisões aleatórias durante o posicionamento. Portanto, o YOTT insere uma metodologia sistemática para transferir a localidade do grafo de entrada para saída considerando as reconvergências e o posicionamento das entradas e saídas nas bordas. O resultado gerado melhorou a qualidade quando comparado com a abordagem YOTO.

3.1 Trabalhos Relacionados

Esta seção apresenta trabalhos relacionados ao problema de P&R com foco em abordagens que exploram teoria de grafos e mapeamento para CGRA.

Em (Ferreira et al., 2005) é apresentado um algoritmo de mapeamento para CGRA assíncrono que percorre grafo por níveis, seguindo a proposta apresentada em (Koren et al., 1988) para posicionamento de grafos em arquitetura de malha. Além de não considerar o escalonamento, pois os operadores são assíncronos sacrificando a vazão

para garantir o correto funcionamento, esta proposta apresenta uma baixa taxa de ocupação da arquitetura alvo e congestionamento de interconexões entre os níveis. A Figura 3.1 ilustra um exemplo de mapeamento por níveis com baixa ocupação e roteamento longos considerando um grafo pequeno e uma arquitetura hexagonal. A Figura 3.1(a) apresenta o grafo de entrada considerando dois operadores dentro de cada nó. A Figura 3.1(b) mostra que os 18 nodos foram mapeados em 29 células hexadecimais usando o algoritmo de (Koren et al., 1988). A ocupação da arquitetura foi de 23% com ALUs, 61% com roteamento e 16% vazias. Sendo 32 células roteadas somente por fios.

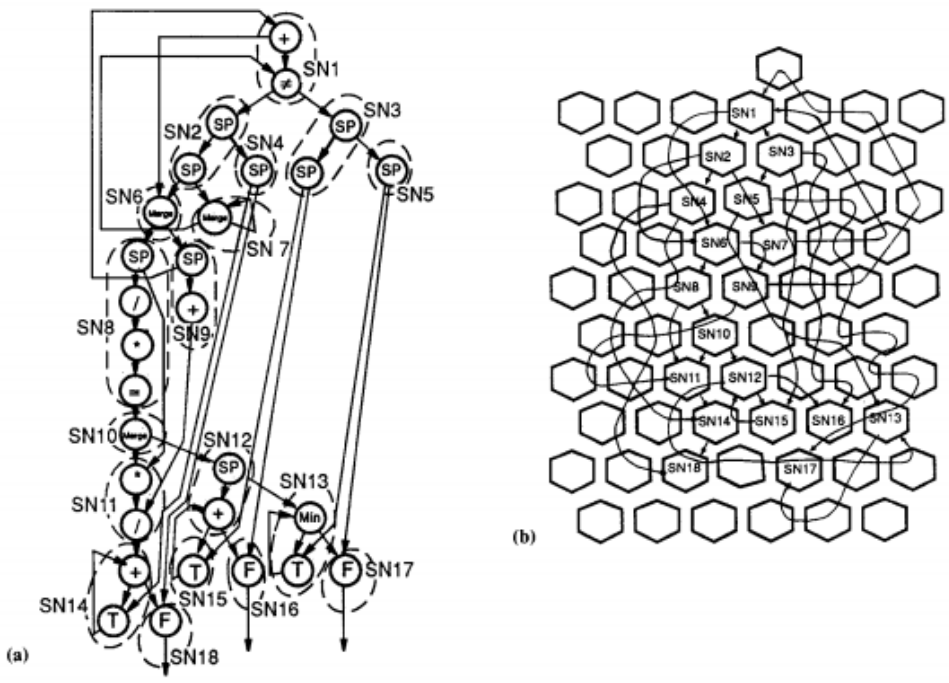

Figura 3.1: Exemplo de mapeamento de grafo por níveis em uma arquitetura hexadecimal. Retirada de (Koren et al., 1988).

Para melhorar a ocupação, (Ferreira et al., 2005) propuseram uma abordagem gulosa para fusão de níveis consecutivos como ilustrado na Figura 3.2. A Figura 3.2(a) mostra um grafo com 13 nodos e 18 arestas que resulta em um primeiro posicionamento por níveis ilustrado na 3.2(b). A Figura 3.2(c) apresenta a fusão da primeira linha com a segunda linha, gerando uma compactação e um reposicionamento do nodo A entre os nodos B e C para diminuir a ocupação no grid. No último nível, uma mudança do nodo M para a posição mais próximo do nodo H também foi realizada. A Figura 3.2(c) ilustra o roteamento final, onde o roteamento que sai dos nodos B, C, D, F, G faz o uso da técnica *multicast*.

Uma abordagem semelhante foi apresentado em (Robič and Šilc, 1994), porém explorando SA para fazer a compactação dos níveis como ilustrado na Figura 3.3. Comparando um grid inicialmente já posicionado e roteado, como ilustrado na Figura

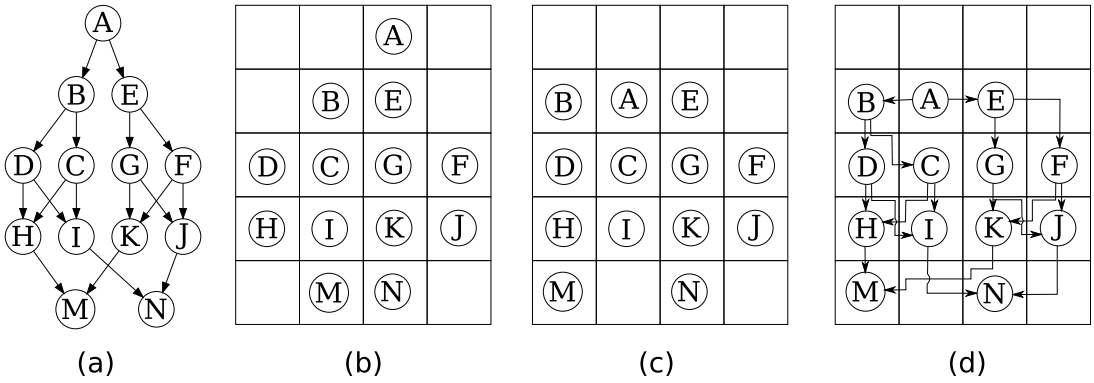

Figura 3.2: Mapeamento utilizando abordagem gulosa. (a) Grafo de entrada; (b) Mapeamento em um grafo por nível e profundidade; (c) Ajustes no mapeamento; (d) Roteamento do grafo. Adaptado de (Silva et al., 2006).

3.3(a), com a aplicação do SA para a diminuição do espaço no grid, pode-se observar uma diminuição de área ocupada passando de um grid de 7x18 para um grid 5x12, ou seja, uma redução de 47% em área, porém a ocupação ainda é baixa.

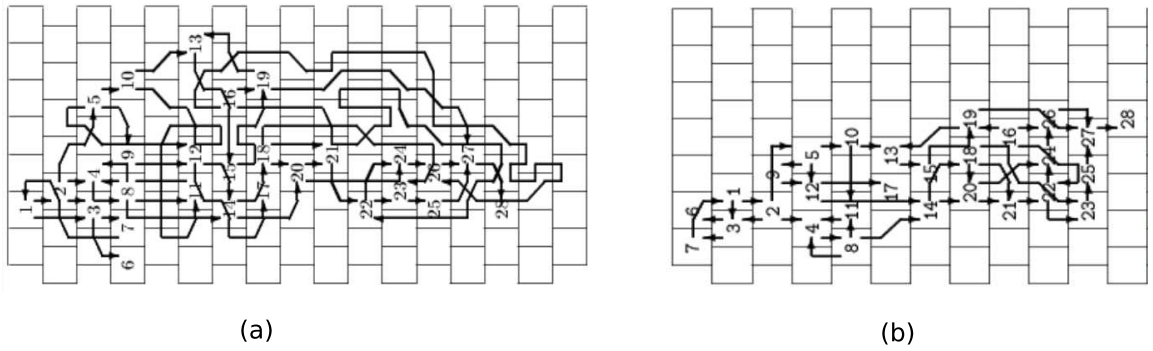

Figura 3.3: Compactação de níveis utilizando *Simulated Annealing*. (a) Grafo antes da compactação; (b) Grafo após a compactação. Adaptado de (Silva et al., 2006).

Considerando o P&R com balanceamento, mas para mapeamento de circuitos em QCA, um algoritmo guloso baseado em níveis foi apresentado em (Trindade et al., 2016). Devido as restrições de recursos de roteamento dos QCA em comparação com os CGRA e a solução gulosa apresentada, a abordagem foi avaliada em grafos com menos de 10 nodos (Trindade et al., 2016) com escalabilidade limitada. Uma solução exata para mapeamento em QCA é proposta em (Fontes Junior et al., 2018), que explora a partição do grafo em subgrafos para depois aplicar o mapeamento por níveis. Grafos com até 30 nodos foram mapeados porém o tempo de execução é da ordem de minutos a horas. As restrições de roteamento nas arquiteturas de QCA reduzem significativamente o número de soluções válidas e a taxa de ocupação das arquiteturas.

Uma abordagem utilizando algoritmo genético para o mapeamento em CGRA é apresentada em (Da Silva et al., 2006). A solução maximiza a ocupação porém

considerada arquiteturas com muitos recursos de roteamento. Para gerar a população inicial, algoritmos de travessia em grafos foram utilizados juntamente com percursos no grafos da arquitetura como ilustrado na Figura 3.4. A Figura 3.4(a) apresenta o grafo de entrada onde foi realizado um percurso em profundidade A, C, E, G, J e I, depois retorna em A para seguir pelo caminho B, D, M, volta em D e percorre F e H. Na arquitetura diversos percursos foram propostos. As Figuras 3.4(b) e (c) ilustram dois exemplos. O primeiro faz um zigzag em linha na arquitetura e segue sequencialmente a lista de profundidade do grafo de entrada. O segundo faz um zigzag em diagonal. Muitas outras variações são feitas gerando a população inicial de P&R para o algoritmo genético otimizar.

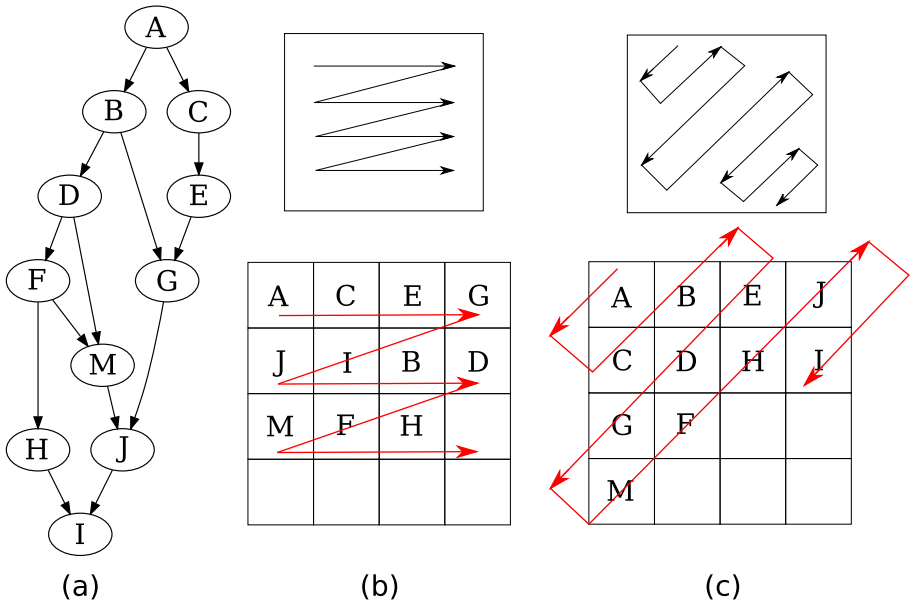

Figura 3.4: Percusso dos grafos. (a) Grafo de fluxo de dados; (b) Busca em profundidade; (c) Busca em amplitude. Adaptado de (Silva et al., 2006).

Um algoritmo de P&R em tempo polinomial $O(|V|^2)$ é proposto em (Lai et al., 2005), onde V é o conjunto de vértices do grafo da aplicação. A proposta é baseada na construção de uma árvore geradora mínima para posteriormente percorrer o grafo e realizar o posicionamento. Entretanto as soluções geradas apresentam baixa taxa de ocupação. Uma solução baseada em algoritmos para desenhos de grafos com a abordagem *split&push* é apresentado em (Yoon et al., 2009a). Por onde a cada passo o grafo vai sendo dividido em partições menores e uma solução usando programação inteira é aplicada, assim como mostrado em trabalhos recentes (Walker and Anderson, 2019b), tem problemas de escalabilidade e tempo de execução da ordem de segundos a minutos para grafos com 20 à 30 nodos.

Duas versões são propostas em (Ferreira et al., 2007) que resolvem o problema P&R em tempo polinomial utilizando a estratégia da travessia em profundidade. A versão do algoritmo mais simples tem uma complexidade $O(|E|)$, onde $|E|$ é a quantidade

de arestas. O algoritmo percorre utilizando busca em profundidade de forma gulosa. A versão do algoritmo mais complexo tem uma complexidade $O(n + |E|^3)$. Nesse algoritmo utiliza-se Dijkstra para guiar o percusso do grafo. A solução desse problema buscou reduzir a ocupação na arquitetura, fazendo com que utiliza-se o mínimo possível de recurso. Entretanto a arquitetura alvo considera um CGRA assíncrono. Caso algum caminho fique desbalanceado, a vazão será reduzida, pois alguns PEs irão aguardar mais tempo até todos os dados chegarem para disparar a computação local.

Uma solução para reduzir o tempo de execução do P&R é adicionar mais recursos de roteamento como, por exemplo, a inclusão de uma rede multiestágio global de interconexão (Ferreira et al., 2009, 2011b). Uma rede multiestágio tem o custo $O(n \log n)$ e algoritmos polinomiais de roteamento. A proposta apresentada em (Ferreira et al., 2009, 2011b) é uma solução híbrida com uma malha e uma rede multiestágio. Cada PE tem o recurso de rotear uma conexão através da rede.

Em (Oliveira et al., 2020; Carvalho et al., 2020) apresentam o uso da heurística *Simulated Annealing* utilizando uma abordagem de troca de posições locais. Em (Oliveira et al., 2020), além de uma de arquitetura homogênea, três arquiteturas heterogêneas (xadrez, coluna e borda) são avaliadas. 1000 soluções são geradas e avaliadas. Os resultados mostram que o tempo de execução 52% mais rápido que a ferramenta VPR 8.1 (Murray et al., 2020) com uma redução de custo sem perda de otimização de 76%. Comparando com o algoritmo YOTT, proposto nesta dissertação, considerando a exploração de 1000 soluções, o trabalho de Oliveira *et al* (Oliveira et al., 2020) reduz em 7% o comprimento de fios e em 15% o tamanho das filas. Entretanto, o tempo de execução é 1000x maior.

Considerando FPGA como arquitetura alvo, (Ferreira et al., 2013b, 2015) aplicam os algoritmos de travessias de grafos propostos em (Ferreira et al., 2007) priorizando o caminho crítico. Os resultados mostram uma redução de três ordens de grandeza no tempo de execução com uma perda de 30% no caminho crítico em comparação com a ferramenta VPR (Luu et al., 2011). Grafos com até 3000 nodos foram avaliados. Entretanto, o mapeamento em FPGA não considera balanceamento dos caminhos.

Outros trabalhos procuram utilizar inteligência artificial para resolver problemas de P&R. Em (Liu et al., 2018) é proposto o algoritmo RLMap para resolver o problema de mapeamentos de grafos em CGRAs como um agente na aprendizagem de reforço, no qual unifica o posicionamento, o roteamento e a inserção de PE por ações do agente. A Figura 3.5(a) apresenta o posicionamento inicial do *benchmark fdtd-apml* de tamanho de 21 nodos e com 22 arestas, mapeados em um *grid* 6×6 com três rotas inválidas $h - k, d - c$ e $u - v$ (arestas em vermelho no posicionamento do grafo). Utilizando a rede de aprendizado *Q-network*, chega-se ao resultado apresentado na Figura 3.5(b). Onde é gerada uma solução ocupando um espaço maior no *grid*. A Figura 3.5(c) apresenta um treinamento da rede após 10 mil passos, chegando a um

resultado ótimo sem problema de conflito de rotas e reduzindo a ocupação. Seus resultados mostraram um desempenho comparável na qualidade com as heurísticas DFGNet (Yin et al., 2017), Pattern (Mehta et al., 2013) e SPKM (Yoon et al., 2008). Além disso, se adapta a diferentes arquiteturas com uma conversão rápida. Contudo, o tempo de execução é elevado. Por exemplo, para o grafo da Figura 3.5 com apenas 21 nodos foram necessários 16 minutos para encontrar a solução.

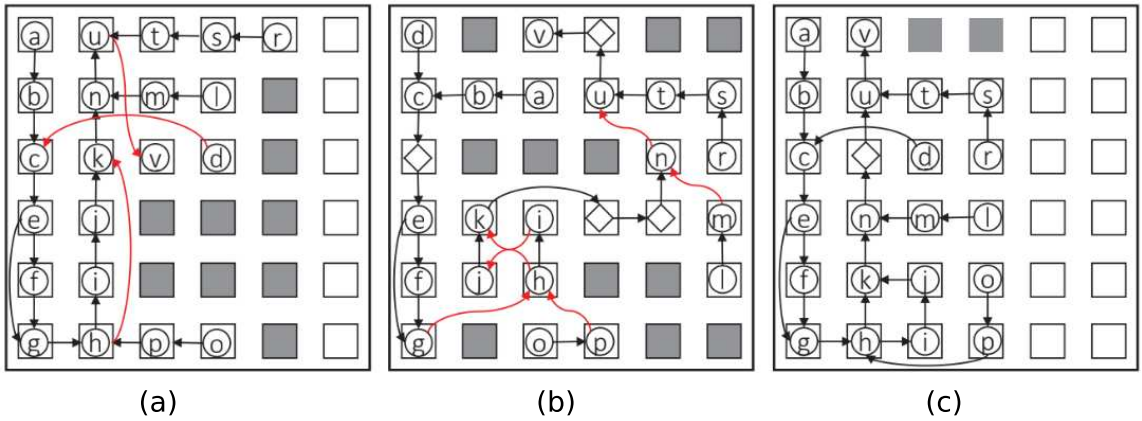

Figura 3.5: Estudo de caso do *Benchmark fdtd-apml*. (a) Posicionamento inicial; (b) Resultado do mapeamento após o primeiro passo; (c) Resultado final do mapeamento. Retirado de (Liu et al., 2018).

Em (Zhao et al., 2020) utiliza o modulo scheduling no mapeamento de grafos de CGRAs. Eles decompõem o problema do mapeamento temporal e espacial em duas etapas. O algoritmo de mapeamento está integrado ao ambiente de compilação LLVM (Lattner and Adve, 2004). Seus resultados mostraram melhora no desempenho de 5.4% a 14.2% dentre os *benchmarks* compilados com sucessos em relação ao algoritmo de P&R RAMP (Dave et al., 2018). Além disso, eles chegaram a utilizar benchmarks com até 154 nodos, mas os seus resultados chegaram a demorar 14 segundos para mapear, inviabilizando em aplicações de tempo real.

Em (Kojima et al., 2020) um algoritmo genético para o mapeamento dinâmico com modulo scheduling em CGRA (GenMap). Os 7 benchmarks utilizados foram de 12 à 45 nodos de tamanho e baseados em aplicações de processamento de sinal e criptografia. Seus resultados mostraram uma redução de até 15.7% em comprimento de fio sem alteração na qualidade da solução comparado com as abordagens SPKM (Yoon et al., 2008), RLMap (Liu et al., 2018), e CGRA-ME (Chin et al., 2017a). GenMap apresenta uma aceleração de 2x, 1.55x e 1.5x em relação as abordagens CGRA-ME, SPKM e RLMap, respectivamente.

3.2 Algoritmo baseado em travessia de grafos

3.2.1 Algoritmos baseados em profundidade

Ferreira et al. (2007) propôs duas versões de algoritmos utilizando a travessia em profundidade. Ambas versões obtém uma solução em tempo polinomial. A versão mais simples percorre o grafo em profundidade de forma gulosa enquanto que a versão mais complexa utiliza Dijkstra para guiar no percurso do grafo. Entretanto a arquitetura alvo considera um CGRA assíncrono. Caso algum caminho fique desbalanceado, a vazão será reduzida, pois alguns PEs irão aguardar mais tempo até todos os dados chegarem para disparar a computação local. A proposta do algoritmo é realizar uma travessia no grafo de fluxo de dados e no grafo que representa a arquitetura alvo. Existem muitas possibilidades para a travessia do grafo de fluxo de dados e do grafo da arquitetura. Ferreira et al. (2007) avalia 1, 10 e 50 travessias aleatórias. Esta dissertação apresenta o algoritmo Traversal que também explora várias travessias de forma eficiente durante o mapeamento. A solução proposta parte de um nó inicial, percorre o grafo utilizando alguma estratégia de travessia (largura, profundidade, ...). Durante o percurso, o algoritmo tem que tomar decisões. Por exemplo, a Figura 5.3(a) apresenta um exemplo de travessia. Primeiro, poderíamos começar pelo nodo A ou pelo nodo E. No exemplo começamos por A. Em A existem duas opções a seguir. Tomamos a decisão de seguir por B. Depois em B temos novamente duas opções, onde seguimos por D. Estas decisões foram realizadas no escopo do grafo de entrada.

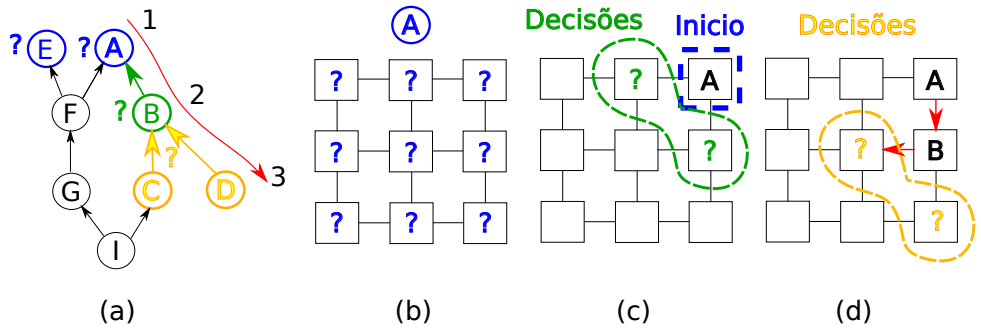

Figura 3.6: (a) Grafo de fluxo de dados; Tomadas de decisão: (b) Nodo inicial; (c) Posicionamento de um novo nodo; (d) Escolha de um novo nodo.

O mapeamento além de percorrer o grafo de entrada, percorre o grafo da arquitetura simultaneamente. Portanto, mais decisões devem ser tomadas ao percorrer o grafo da arquitetura. A Figura 5.3(b) ilustra 9 possibilidades para posicionar o nodo A que é o primeiro nodo percorrido do grafo de entrada. Uma vez posicionado A, a Figura 5.3(c) mostra a próxima decisão a ser tomada. Neste caso, o algoritmo explora a localidade do grafo de entrada e saída. Como B é vizinho de A no grafo de entrada, o algoritmo irá posicionar B em um nodo vizinho de A no grafo da arquitetura. Im-

portante destacar que esta operação é a base para o sucesso do algoritmo de travessia. O nodo A já se encontra posicionado e estamos visitando B através da aresta $A \leftarrow B$. Portanto, A irá determinar a posição de B buscando uma aresta no grafo da arquitetura onde A e B sejam vizinhos, transferindo assim a localidade de um grafo para o outro. Neste caso temos duas opções. Suponha que B seja posicionado abaixo de A como ilustrado na Figura 5.3(d). Agora o próximo passo irá buscar uma posição para o nodo D como vizinho de B com duas opções. A qualidade da solução dependente diretamente das decisões tomadas durante a travessia do grafo de entrada e do grafo da arquitetura.

3.2.2 Algoritmo YOTO

No artigo apresentado no Capítulo 2 é abordado o algoritmo para percurso Zigzag (Seção 4.3.3). Diferente dos algoritmos tradicionais, Zigzag percorre tanto na direção saída para entrada (S->E) tanto no sentido entrada para saída (E->S) buscando capturar as correlações em grafos com múltiplas saídas. A figura 3.7 compara uma travessia em profundidade com uma travessia com Zigzag. A Figura 3.7(a) apresenta o primeiro caminho, partindo da saída A, em profundidade na seguinte ordem: A, C, G, E, H, onde todos os descendentes de A foram percorridos. Porém em um grafo com múltiplas saídas, temos que recomeçar a travessia no nodo B para percorrer todo o grafo. Neste exemplo, o caminho percorrido foi B, D, F. Note que o algoritmo não observa correlações entre as saídas A e B do grafo. Neste exemplo, as arestas $D \leftarrow E$ e $F \leftarrow H$ podem requerer vários segmentos, pois o caminho que parte de B não tem nenhuma informação do posicionamento de E e H que foram realizados durante o caminho que começou na saída A. A Figura 3.7(b) mostra uma solução de P&R para uma travessia no grafo da arquitetura, seguindo a ordem em profundidade destacada com a linha pontilhada no grafo de entrada apresentado na Figura 3.7(a). Podemos observar que as arestas que foram visitadas tiveram sua localidade transferidas para o grafo de saída. Porém as arestas $D \leftarrow E$ e $F \leftarrow H$ tiveram um custo maior pois não foram visitadas, já que os nodos E e H já estavam posicionados na arquitetura quando D e F foram visitados. Além das múltiplas saídas, temos as reconvergências internas. Neste exemplo, a aresta $E \leftarrow G$ também não foi considerada, pois G já tinha sido posicionado quando E foi visitado. Resumindo, todas as arestas que não foram consideradas podem gerar um custo maior em segmentos pois a sua informação de localidade não foi avaliada no mapeamento. A Figura 5.3 mostra com uma linha cinza pontilhada o percurso de travessia no grafo de entrada e da arquitetura.

O percurso Zigzag é apresentado na Figura 3.7(c). A mudança de sentido ocorre quando o número de arestas de um nó é maior ou igual a 2 para as entradas (fanin) ou

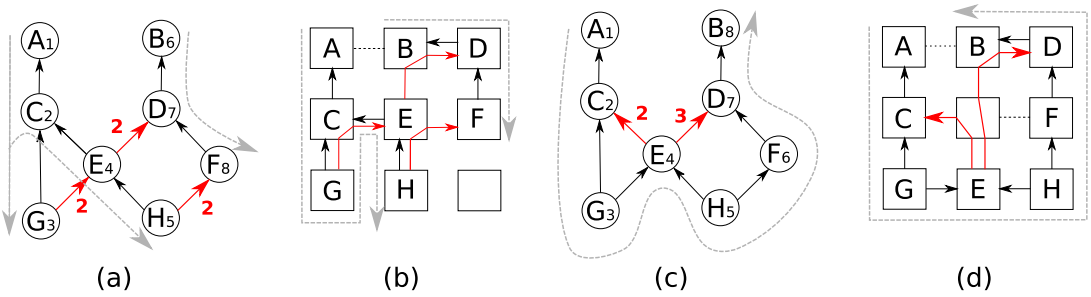

Figura 3.7: Comparação do Percurso em Profundidade com o Percurso ZigZag: (a) Grafo de Entrada com Travessia em Profundidade; (b) Grafo da Arquitetura em Profundidade; (c) Travessia em ZigZag do Grafo de Entrada; (d) Grafo da Arquitetura percorrido em ZigZag.

para as saídas (fanout). Desta maneira, quando iniciamos pelo caminho A, C, G não ocorre inversão de sentido no percurso, pois o nodo C possui um fanout 1 e a direção do percurso é saída para entrada. A direção é alterada quando o fanout é maior do que 1, o que acontece quando o nodo G é visitado. O sentido é invertido, agora indo das entradas para as saídas. O sentido será novamente alterado quando chegar no nodo E que possui fanin igual a 2. Esse processo continua até todos os nodos serem visitados. O percurso final para o exemplo foi: $A_1, C_2, G_3, E_4, H_5, F_6, D_7, B_8$. A Figura 3.7(d) apresenta o grafo da arquitetura que também foi percorrido nesta sequência. O resultado final depende também das decisões que foram tomadas durante a travessia da arquitetura. Em comparação com a travessia em profundidade para este exemplo, a travessia Zigzag visitou uma aresta a mais e explorou sua localidade. Porém duas arestas não foram exploradas: $C \leftarrow E$ e $D \leftarrow E$. Apesar da travessia Zigzag resolver as correlações das múltiplas saídas, o problema de reconvergências ainda continua presente. As informações de localidade destas arestas não são transferidas para o percurso do grafo da arquitetura, resultando em arestas com vários segmentos. Na próxima seção apresentamos o algoritmo YOTT que realiza duas travessias no grafo de entrada para capturar e transferir esta localidade.

3.3 Algoritmo YOTT

Algoritmos que abordam a estrategia de travessia em um grafo irão tomar decisões durante o percurso (como discuto na subseção 3.2). Os algoritmos de travessia propostos na literatura (Lai et al., 2005; Ferreira et al., 2007; Canesche et al., 2020) percorrem uma única vez o grafo (YOTO - *You only traversal once*). Para maioria das arestas (50 a 90%), a localidade do grafo de entrada é transferida para o grafo da arquitetura. Porem para as arestas não visitadas as decisões durante o percurso são gulosas e de forma aleatória. Portanto não consideram os caminhos reconvergentes. Nesta dissertação introduzimos uma nova abordagem, o algoritmo YOTT (*You only traversal*

twice), onde executamos duas passagens no grafo de entrada e uma no grafo de saída. A primeira travessia coleta as informações para detectar os caminhos reconvergentes e faz anotações. A segunda visita percorre novamente o grafo de entrada e ao mesmo tempo irá percorrer o grafo da arquitetura utilizando as anotações feitas na primeira travessia para guiar algumas decisões no percurso do grafo da arquitetura. Na ausência de anotações, decisões gulosas serão executadas.

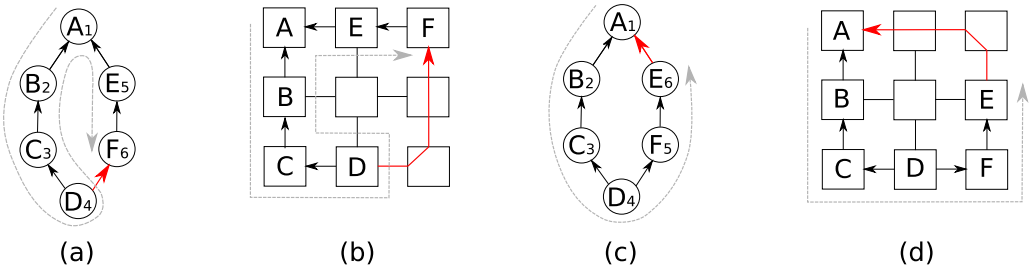

Figura 3.8: Reconvergência em grafos de fluxo de dados: (a) Percurso em Profundidade; (b) Uma solução no grafo da arquitetura em profundidade; (c) Percurso em Zigzag; (d) Mapeamento na arquitetura alvo para o percurso em Zigzag.

As reconvergências geram problemas em vários algoritmos de busca. A Figura 3.8 exemplifica com dois caminhos. A Figura 3.8(a) ilustra o caminho percorrido utilizando algoritmo de profundidade na seguinte ordem: A_1, B_2, C_3 e D_4, depois retornamos ao nodo A para continuar na ordem E_5 e F_6. A Figura 3.8(b) mostra uma solução para grafo mapeado. Observe que a aresta $F \leftarrow D$ não é considerada na travessia e gera um caminho longo. A Figura 3.8(c) apresenta o caminho percorrido utilizando algoritmo de Zizag com a sequência: $A_1, B_2, C_3, D_4, F_5, E_6$. A Figura 3.8(d) mostra uma solução para grafo mapeado com Zigzag. Novamente, uma aresta que gera a reconvergência não é visitada e pode resultar em um custo alto. Em ambos algoritmos, o problema da reconvergência ocorre e foi destacado com a cor vermelha as arestas na Figura 3.8.

O algoritmo YOTT, apresentado no Capítulo 4, propõe uma solução para capturar a localidade das reconvergências. A Figura 3.9(a) mostra um grafo de entrada que foi percorrido utilizando o algoritmo Zigzag. A ideia é adicionar uma lista de anotações (LA) durante a primeira vez que o grafo de entrada é percorrido. O percurso pode ser largura, profundidade, zigzag, etc. Todas as arestas serão percorridas e a lista é dinamicamente gerada nos momentos que as reconvergências são detectadas. Isto ocorre quando um nó é visitado pela segunda vez. Neste momento caminhará na sequencia já percorrida no sentido inverso para realizar as anotações.

Primeiro ponto a ser relembrado: a base do algoritmo de travessia é transferir a localidade de uma aresta $A \leftarrow B$, onde A já está posicionado e irá buscar uma posição para B próximo de A. Resumindo a posição de B é definida a partir da posição de A. A presença de uma reconvergência em B, por exemplo resultante da aresta $C \leftarrow B$,

implica que terá que buscar uma posição para B que seja perto de A e de C. Como A e B já foram visitados, não se tem nenhum controle ao posicionar C e não se sabe se C irá ficar perto de B. Porém ao visitar C já será conhecido onde A e B estão na arquitetura alvo, pois foram visitados antes. Então ao detectar a reconvergência $C \leftarrow B$, pode-se guiar o nodo C e seus antecessores na direção de B.

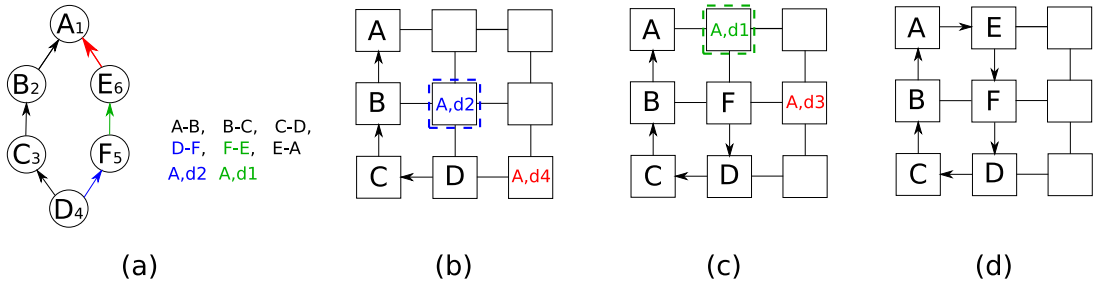

Figura 3.9: Algoritmo YOTT: (a) Grafo com anotações; (b) Posicionamento de F seguindo as anotações; (c) Posicionamento de E; (d) Solução ótima resultante das anotações do YOTT.

A Figura 3.9(a) mostra uma reconvergência que ocorre no nodo A quando a aresta $E \rightarrow A$ é visitada. A posição de A já é conhecida. Será anotado nas arestas anteriores a aresta $E \rightarrow A$ que E deve ficar perto de A. Primeiro, na aresta $F \rightarrow E$ que determina a posição de E, anota-se que E deve ficar a uma distância 1 de A, além de ter que satisfazer a localidade de estar a uma distância 1 de F que determina sua posição. Esta informação é propagada para as arestas anteriores, onde F terá que estar no máximo a uma distância 2 de A, para posicionar E a uma distância 1. As anotações irão ser exploradas na segunda travessia para guiar as decisões no grafo da arquitetura. A Figura 3.9(b) mostra que quando o nodo F será posicionado pela a aresta $D \rightarrow F$, sabemos que além de ser vizinho a D, F deve ir na direção de A e escolher preferencialmente uma distância 2 de A seguindo a anotação. Na próxima aresta $F \rightarrow E$ será usado novamente a anotação para guiar E na direção de A como ilustrado na Figura 3.9(c). Finalmente, a Figura 3.9(d) apresenta o posicionamento e roteamento resultante que é ótimo para o grafo da Figura 3.9(a).

Capítulo 4

TRAVERSAL: A Fast and Adaptive Graph-based Placement and Routing for CGRAs

4.1 Introduction

Over the last decades, data-processing algorithms research led to the investigation of different architecture paradigms ranging from general-purpose solutions to domain-specific hardware with the common goal of improving performance and energy efficiency. This work focuses on streaming-data, which appears commonly in the context of internet-of-things (IoT), digital signal processing (including audio/video), compressed and uncompressed data, and encryption. At the front-end side, a streaming language is a natural way to design a streaming algorithm. There are several streaming languages such as StreamIt (Thies et al., 2002), targeting multi-core architectures (Taylor et al., 2002), OpenSPL (Becker et al., 2016), targeting an application-specific, bit-level architecture on FPGA, and stream-dataflow directives in C Language to attach stream hardware accelerators at the word level (Weng et al., 2019).

At the back-end side, regular architectures are attractive as they can quickly scale to take advantage of more silicon area. A simple and popular architecture is a set of processing elements (PEs) arranged in a 2D fashion as a mesh. The PE granularity ranges from processor core as 16-tile RAW processor (Taylor et al., 2002) at task level to a word-level of functional units as Coarse Grain Reconfigurable Architectures (CGRAs) proposed in (Weng et al., 2019; Mei et al., 2003; Hamzeh et al., 2013; Cong et al., 2014; Ferreira et al., 2013a).

In this work, we focus on the instruction level and hardware accelerators implemented by using CGRAs, where compiler frameworks (Weng et al., 2019) extract a dataflow graph. In this dataflow graph, each node is a primary arithmetic operation, such as addition or multiplication. The compiler framework also decouples the main memory to generate stream data that comes from/to scratchpad memories attached to the accelerator architecture. Our challenge is how to map the dataflow graph onto the hardware accelerator efficiently.

CGRAs are a hybrid computational architecture that benefits from the parallel

customizability of low-level logic systems, such as FPGAs, but CGRAs are a coarser level of design abstraction that makes them easier to design for and quicker to configure in reconfigurable systems. For several applications, CGRAs can improve the run-time in what is called reconfigurable computing (Liu et al., 2019b). Compared to standard FPGAs, CGRAs possess more complex processing elements (PEs), which are programmable at the word-level instead of the bit-level. These PEs enable CGRAs to be quickly configured and make the systems easier to design for compared to the low-level bit-configuring of logic operations on FPGAs. Similarly, the configurable customization that is possible on CGRAs makes them more efficient, for some applications, compared to both general-purpose processors and stream-based processors (Taylor et al., 2002) since the PEs are simple and customized to the application.

A CGRAs could execute designs described as dataflow graphs, which is mapped to the architecture as part of the CAD flow, and one step in this flow is mapping nodes of a dataflow design graph to the PEs of the CGRA, called placement and routing (P&R). P&R algorithms determine where to place the computation nodes onto physical PEs and how to interconnect them. The P&R problem for CGRAs is similar to that of P&R for FPGAs and VLSI.

Although P&R is an NP-Complete problems, P&R for CGRAs maps designs onto fewer PEs making the instances of this problem smaller and less complex. Thus, there are new possible heuristic approaches that may work well in this specific space. However, P&R for CGRAs includes additional mapping constraints that include balancing the latency on the routing paths and modulo scheduling for loop operations. This makes P&R for CGRAs an interesting and challenging problem that is difficult to be solved (Nowatzki et al., 2018b). Even for small instances, recent exact approaches based on Integer Linear Programming (ILP) (Chin and Anderson, 2018a; Yoon et al., 2009b), SAT solvers (Donovick et al., 2019), and hybrid approaches (ILP+heuristics) (Nowatzki et al., 2018b) require minutes to perform the P&R. This work focuses on examining how Simulated annealing (Mei et al., 2003) and graph-based heuristics (Lai et al., 2005; Ferreira et al., 2007) can reduce the P&R execution time.

Specifically, this work introduces and explores an algorithm that maps a dataflow input graph onto a CGRA architecture by exploiting graph traversal techniques where both the design and the architecture of the CGRA are represented as graphs, and the concurrent traversal of both graphs transfers the locality from the input design graph to the target architecture graph. We explore traditional graph traversals such as breadth-first search and depth-first search, and we introduce a novel zigzag traversal to explore the correlation between multiple outputs. We define the P&R quality based on the number of buffers or delay FIFOs (Nowatzki et al., 2018b). Our P&R tries to balance the computation branches, considering how the input and target graphs are

traversed and mapped to the architecture.

Our algorithms are polynomial greedy approaches that execute quickly. Because of this fast execution, we propose to execute several instances of this algorithm, where each instance makes different traversal choices at the input and target graphs to explore the solution P&R space. Additionally, we show how to implement a parallel version of this approach on a GPU to improve overall execution time.

We compare our traversal approach with a state-of-the-art CGRA-ME framework, which implements simulated annealing (SA) and integer linear programming (ILP) solvers for CGRA P&R. Furthermore, we show how our approach can generate higher quality mappings and improve the execution time for a set of benchmarks by at least five orders of magnitude. Finally, considering spatial mapping at a millisecond scale, we compare our approach to the same implementation on the VPR tool achieving a speed-up factor of $10\times$.

We organize the paper as follows. In section 4.2, we describe details about CGRAs, P&R, and existing P&R algorithms in the CGRA space. In section 4.3, we describe our graph traversal P&R algorithm and present execution examples. Next, in section 4.4, we show how to parallelize our base kernel algorithm on a GPU. In section 4.5, we compare our algorithm to existing state-of-the-art algorithms. Finally, in section 4.6, we conclude the paper and provide directions for future work.

4.2 Background

4.2.1 Coarse-Grained Reconfigurable Architecture

CGRA architectures and tools have been studied over the last three decades (Liu et al., 2019b). A CGRA mitigates some of the FPGA shortcomings by shifting reconfigurable granularity from the bit-level to the word-level resulting in a time reduction for mapping to the less complex architecture. The main disadvantages of CGRAs are the low number of commercial architectures and design toolchains (Liu et al., 2019b).

CGRA architectures can be mesh-based (Mei et al., 2003), stripe-based (Mehta et al., 2015), and island-based (Coole and Stitt, 2010). Mesh-based architectures are by far the most popular because these CGRAs are scalable, and have low local interconnection costs. Figure 4.1(a) shows a 4x4 mesh CGRA. Bansal (Bansal et al., 2004b) show that adding interconnections with 1-hop, it is enough to exploit the available instruction-level parallelism. Figure 4.1(b) depicts this 1-hop pattern where a $PE_{i,j}$ connects to eight adjacent nodes, two in each direction (east, west, north, and south). Considering the 1-hop interconnection, Figure 4.1(c) shows the short distance from the highlighted cell $PE_{2,2}$ to all other cells in 4x4 mesh CGRA, where there are 6 direct connections (cost= 0) and 9 PEs with distance 1. Figure 4.1(d) displays an

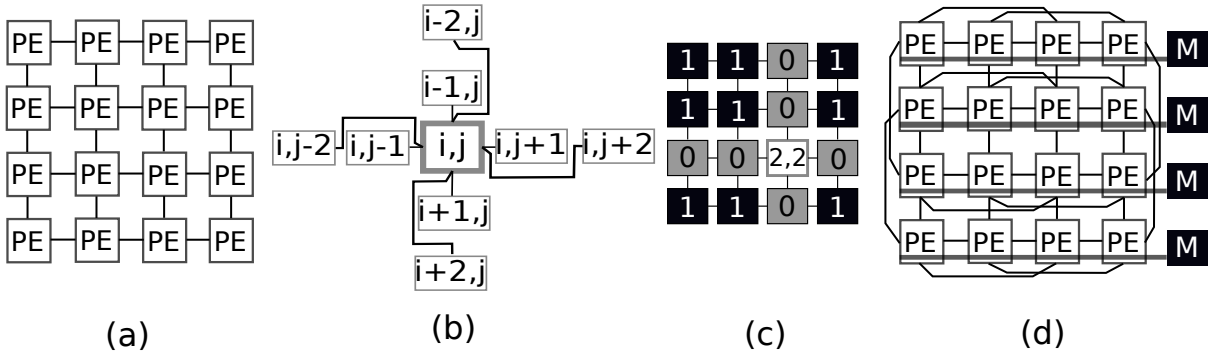

Figura 4.1: (a) A 4x4 mesh-based CGRA; (b) 1-hop connection; (c) Short distance; (d) ADRES CGRA 1-hop plus one memory per row.

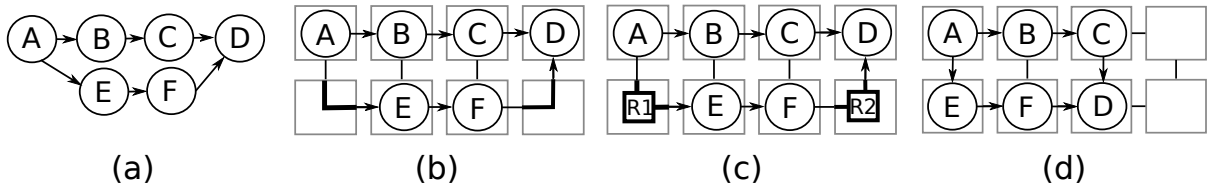

Figura 4.2: (a) A dataflow graph; (b) P&R with unbalanced highlighted wires; (c) Unbalanced due Routing Registers; (d) Balanced P&R.

example of a popular mesh architecture, the ADRES CGRA (Mei et al., 2003), with 1-hop connections and one memory module per row. All PEs in the same row have direct and not concurrent memory access since they use the same bus connection.

4.2.2 Latency Balancing during P&R

CGRAs can exploit both spatial and temporal parallelism. For stream computation, a correct P&R provides the same latency on all paths that converge to a given node. Therefore, P&R algorithms need to perform "latency balancing" at the node granularity. Balancing paths fixes delay mismatches between two paths due to different arrival times at reconverging nodes.

Figure 4.2(a) and (b) shows a dataflow and a P&R solution. In this case, the P&R solution does not satisfy the latency balancing constraints (see Figure 4.2(c) since the path $A \rightarrow r_1 \rightarrow E \rightarrow F \rightarrow r_2 \rightarrow D$ is longer than the path $A \rightarrow B \rightarrow C \rightarrow D$, due the routing registers r_1 and r_2). Figure 4.2(d) depicts a feasible P&R solution where all paths have the same latency, and every wire segment has a register to preserve the pipeline clock.

4.2.3 Modulo Scheduling

Modulo Scheduling (MS) is a software pipelining approach that solves loop overlapping execution by directly generating a schedule for a single iteration that repeats for

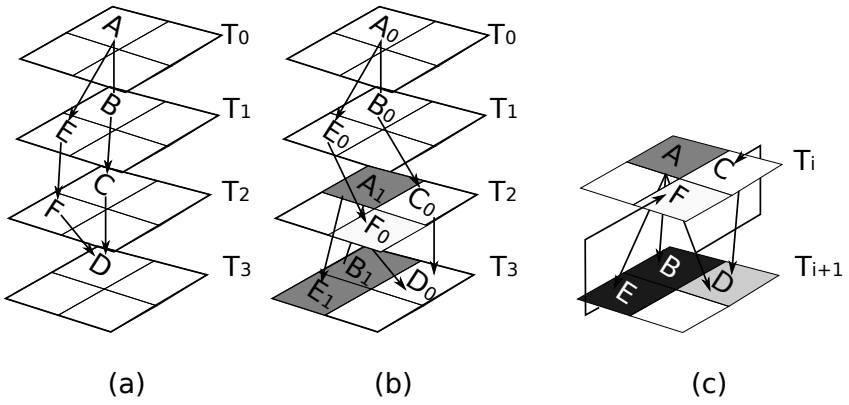

Figura 4.3: (a) A latency= 4 and Initiation Interval= 4; (b) Initiation Interval= 2; (c) Modulo Scheduling $II = 2$, and two overlapping iterations.

a fixed number of cycles (Rau, 1996). When the dataflow has more node operators than processing elements in the target architecture, the computation is done by using time-multiplexed approaches (time partitions).

As an example, assume you have a dataflow of six operations to be mapped on a 2×2 PEs mesh CGRA. Figure 4.3(a) shows scheduling and P&R for the dataflow from Figure 4.2(a). The latency is 4 cycles, and every 4 cycles, a new iteration of the loop completes, *i.e.* the initiation interval $II = 4$. A 2×2 CGRA can perform 4 operations per cycle, but in our example, it only executes 1.5 operations per cycle. A different P&R and scheduling solution, as shown in Figure 4.3(b), has an initiation interval of $II = 2$. At time T_0, the first iteration begins, and at time T_2, the second iteration begins by processing the node A_1 while the first iteration is computing C_0 and F_0. At time T_3, the first iteration finishes by computing D_0, the second iteration is computing B_1 and F_1, and therefore 3 operations per cycle are executed in time T_2 and T_3, achieving a PE occupation of $\frac{3}{4} = 75\%$. Figure 4.3(c) depicts the final modulo scheduling with $II = 2$ that exploits spatial and temporal parallelism, where it is possible to overlap two iterations by using: dark grey and black for the first iteration, and white and light gray for the second.

4.2.4 Previous CGRA P&R Research

Several works propose exact approaches to both P&R and MS based on ILP and SAT methods, e.g. (Nowatzki et al., 2018b; Chin and Anderson, 2018a; Yoon et al., 2009b; Donovick et al., 2019; Walker and Anderson, 2019a). These approaches can find an optimal solution for several CGRA target architectures including interconnection topologies such as 1-hop, diagonal, and toroidal.

Yoon *et al.* (Yoon et al., 2009b) present and evaluate an ILP P&R approach on a set of random dataflows that are 5 to 16 nodes large, where the average algorithm runtime is 13.3 minutes. More recently, Chin and Anderson (Chin and Anderson,

2018a) present their ILP approach that reaches feasible mappings even when the SA algorithm fails. Nevertheless, the mapping execution time can be more than a few minutes for small dataflow graphs with less than 20 nodes. The ILP approach presented in in (Walker and Anderson, 2019a) reduces the execution time by one to two orders of magnitude being able to run in seconds. Nowatzki *et al.* (Chen et al., 2016) propose a hybrid approach that overlaps ILP and breadth-first random heuristic to reduce execution time. Their experiments show that by increasing the area from 10% to 50% with local delay FIFOs, it is possible to reduce the problem "difficulty" for small graphs.

In contrast to the above approaches, graph-based heuristics can improve runtime significantly. Yoon *et al.* (Yoon et al., 2009b) propose a split push kernel mapping (SPKM) based on graph drawing algorithms, where the execution time is four orders of magnitude smaller than their ILP approach. However, SPKM does not support modulo scheduling and performs static spatial mapping. Moreover, the graph drawing approaches increase the minimum area required to reach a feasible solution. Lai (Lai et al., 2005) present another graph-based approach inspired by the Kruskal minimum spanning tree algorithm. Although the placement complexity is $O(n^2)$ for a graph with n nodes, the approach requires more mesh area and does not perform latency balancing nor modulo scheduling.

Ferreira (Ferreira et al., 2007) propose a polynomial graph-based static spatial P&R for CGRAs and FPGAs (Ferreira et al., 2013c). This greedy approach traverses the input graph in depth-first order, and the target architecture performs a one-shot placement quickly. However, this approach does not perform latency balancing nor modulo scheduling. Chen and Mitra (Chen and Mitra, 2014a) propose a graph-based approach, which formalizes the P&R as a graph minor containment problem. The execution time ranges from 10 ms to 100 ms for dataflow sizes ranging from 17 up to 87 nodes, where the target architecture is a 4×4 mesh. For a large CGRA of 16×16 PEs, the average execution time is around 7 seconds.

Friedman (Friedman et al., 2009) propose an adaptive approach to support a variety of CGRAs named SPR (Schedule, Place, and Route). SPR, as other CGRA P&R, uses SA placement (Mei et al., 2003; Hamzeh et al., 2013), and QuickRoute to solve the latency balancing problem (Friedman et al., 2009).

Mehta (Mehta et al., 2015) propose a P&R for a stripe-based CGRA, which is also based on graph drawing algorithms. Although the execution times is two orders of magnitude better than SA approaches, the algorithm achieves this time reduction by increasing the CGRA area with low PE occupation, and a customized row×column CGRA for each dataflow. For instance, a dataflow with 62 nodes requires 17×8 CGRA with 136 PEs, while another dataflow with 32 nodes requires a 9×11 with 99 PEs.

Recently, Liu (Liu et al., 2019a) propose a deep reinforcement learning approach

for spatial CGRA mapping, including latency balancing, but there is no support for modulo scheduling. The execution time is still a challenge where moderate dataflow sizes of 30 nodes require more than 10 minutes to map.

Our work differs from the literature in the following aspects: (a) Our approach is a graph traversal CGRA placement similar to those proposed in (Lai et al., 2005; Ferreira et al., 2007). However, our approach includes modulo scheduling, path balancing, and a novel graph traversal strategy; (b) Our solution is a generic algorithm that can map to different CGRA architectures and topologies; (c) We parallelize our approach using a GPU.

4.3 Traversal Placement

This section presents formal definitions and examples to introduce our traversal-based P&R. We organize as follows: Section 4.3.1 provides formal definitions of the problem; section 4.3.2 describes the placement algorithm, section 4.3.3 introduces a novel traversal strategy; section 4.3.4 details local optimization strategies; section 4.3.5 presents how to model different topologies; section 4.3.6 shows how we include modulo scheduling, and finally in section 4.3.7, we simplify the P&R constraints by using elastic size FIFOs in the PE's local interconnections.

4.3.1 Definitions for the Problem

A data-flow is a directed graph (V, E), where V is the set of nodes and E is the set of edges, where $E \in V \times V$. Each edge e is a tuple (s, d) or $s \rightarrow d$, where s is the edge source node and d is the edge destination node. A node v is an output if and only if it never appears as source node in E. An **input traversal set (ITS)** is a set of edges $ITS \in V \times V$, where $|ITS| \leq |V|$ and $ITS \subseteq E$. Moreover, considering a node $v \in V$ that is not an output, it can appear just once as source node in ITS. Therefore, for all edges source/destination $s_i \rightarrow d_i \in ITS$, if $i \neq j$ then $s_i \neq s_j$.

An input traversal list (ITL) is an ordered sequence of edges containing the elements of ITS. Therefore, ITL provides an order to the elements of ITS.

For the given dataflow graph, there are $|V|!$ possible input traversals. Previous P&R algorithms generate traversal lists based on depth-first order (Ferreira et al., 2007, 2013c), breadth-first order (Nowatzki et al., 2018b), and spanning trees (Lai et al., 2005). These sequences capture the dataflow graph locality where consecutive edges in ITL are neighbors in the original graph. The depth-first traversal approach (Ferreira et al., 2007, 2013c) has a polynomial time complexity, and the ITL is visited once during the placement step. The placement of a node v occurs when the algorithm visits a node during the ITL traversal (Ferreira et al., 2007, 2013c). For instance, suppose the

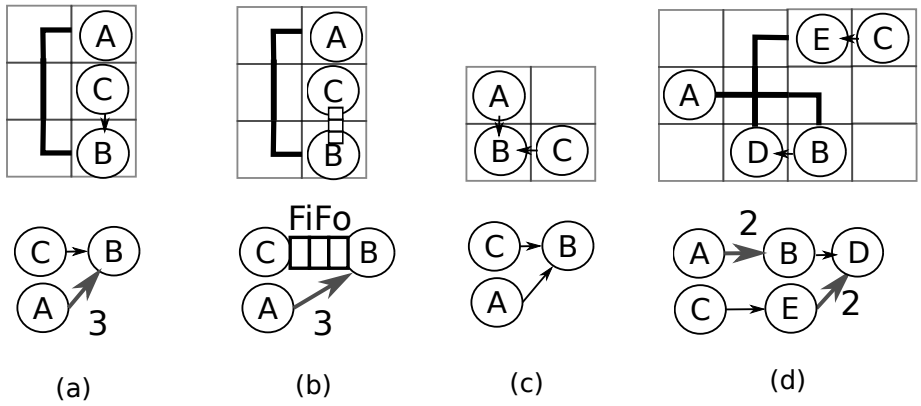

Figura 4.4: (a) Unbalanced P&R; (b) FIFO Balanced P&R; (c) Optimal P&R; (d) Balanced P&R without FIFOs.

$ITL = \{c \rightarrow d, b \rightarrow c, \ldots\}$ for the dataflow shown in Figure 4.2(a). First, we visit the edge $e_1 = c \rightarrow d$. The placement of the node c will be defined by the position of the node d, where c will be placed as close as possible to d. Then, next edge $e_2 = b \rightarrow c$ is visited and b will be placed as close as possible to c. In this way, the placement captures the graph locality. The only exceptions are the output nodes, such as d, that should be placed before the ITL is processed since an output will never be an edge source node.

The target architecture is defined as follows: A target architecture (PE, I) is a graph where PE is the set of processing nodes, and I is the set of edges connecting them. A pe_i and $pe_j \subseteq PE$ are adjacent (or neighbours) if there is a interconnection $I_{ij} = pe_i \rightarrow pe_j$. In addition, the placement function $p : V \rightarrow PE$ maps a dataflow node $v \in V$ onto a $pe \in PE$.

An optimal P&R is defined as follows: A P&R is optimal, if there is a function $pr : E \rightarrow I$, where for all edges $e = (s_i, d_i) \in E$, there is a single interconnection link $w = (pe_k, pe_l) \in I$ such that $p(s_i) = pe_k$, and $p(d_i) = pe_l$.

Figure 4.2(d) depicts an optimal P&R on a mesh architecture, where all edges are mapped with adjacent connections. However, a P&R can be legal even it is not optimal, where at least for one edge $e = (s_i, d_i) \in E$ that $(p(s_i), p(d_i)) \notin I$. In addition, there is a routing path $r = p(s_i) \rightarrow pe_{L_1} \ldots \rightarrow pe_{L_N} \rightarrow p(d_i)$. Therefore, there is at least one edge where source and destination nodes are placed in two non adjacent PEs. Figure 4.2(c) shows a feasible P&R where the edge $A \rightarrow E$ has the interconnection path $p(A) \rightarrow p(r_1) \rightarrow p(E)$ and the edge $F \rightarrow D$ has the interconnection path $p(F) \rightarrow p(r_2) \rightarrow p(D)$.

In a fully pipelined implementation, there is an additional constraint: all paths should be balanced. We assume that every PE has the latency of one unit. A PE could perform a computation (add, multiply, etc), a routing action as r_1 and r_2 in Figure 4.2(c), or both. For ease of explanation, suppose we have the dataflow shown

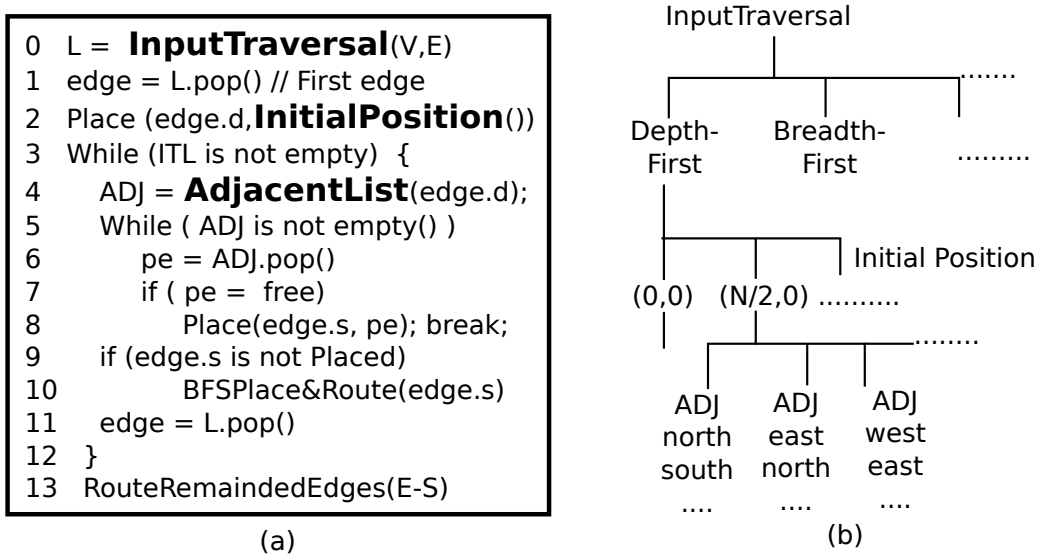

Figura 4.5: (a) P&R Pseudo-code (b) P&R Greedy Instances Space Parameters.

in Figure 4.4(a). The edge $a \rightarrow b$ has the length 3. A FIFO with length 3 should be inserted in the path c to b to avoid a delay mismatch (Nowatzki et al., 2018b), as shown in Figure 4.4(b). Nevertheless, if the algorithm finds an optimal P&R, no FIFO or long wires are required, as shown in Figure 4.4(c). Therefore, in addition to performing the P&R, a timing or FIFO scheduling is required for a non-optimal P&R.

It is possible to perform the placement, routing, and FIFO scheduling independently or simultaneously, and a simple solution is to decouple these steps. However, a feasible routing solution might not exist due to placement, and even when a possible routing exists, the timing solution might fail if there are insufficient FIFOs to avoid the delay mismatches. In (Nowatzki et al., 2018b), an evaluation of the execution of three steps suggests that the routing and the timing should be performed simultaneously. Our approach shows that it is possible to execute it independently to reduce the implementation complexity and find feasible solutions.

Our algorithm uses routing resources as a primary cost metric. Although this can generate non-optimal solutions with long wires, no FIFOs may be required, as shown in Figure 4.4(d), where the long wire of another path balances the long wire from one path. Therefore, the timing step uses a global view of all paths to place and to optimize the FIFO sizes. One contribution of this work shows that it is possible to reach feasible routing and timing solutions by using simple local strategies, as described in the following sections.

4.3.2 P&R Algorithm

Figure 4.5(a) shows the P&R pseudo-code. First, we explain the ITL and the adjacent and non-adjacent placement. Next, we present the routing of the remaining edges,

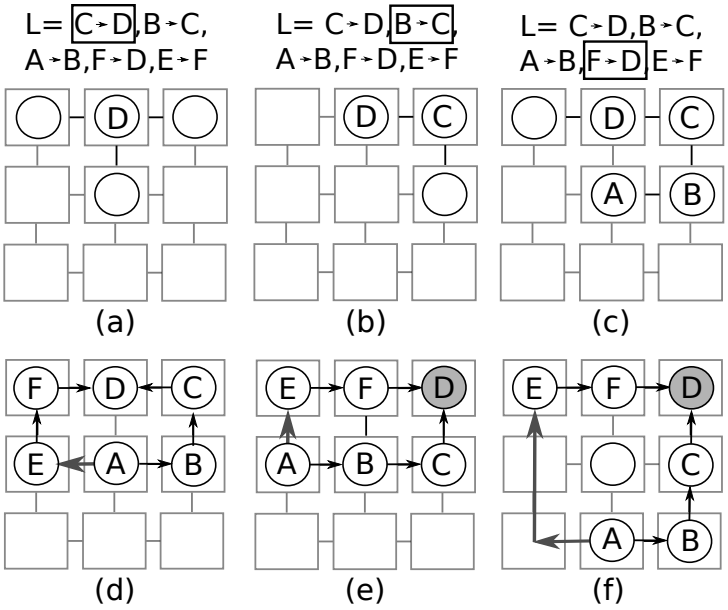

Figura 4.6: (a) Traversal list L and an initial position for d; (b) Place c; (c) Place b and a; (d) Final P&R; (e) New initial position for d; (f) A feasible P&R.

and finally, the solution space exploration.

Input Traversal List P&R

The placement algorithm scans the ordered input traversal list (ITL). For ease of explanation, let us suppose d_i is placed at $pe_j = p(d_i)$ (line 2 in pseudo-code), and $ADJ = \{pe_k | pe_k \rightarrow pe_j \in I\}$ (line 4). For the edge $s_i \rightarrow d_i \in ITL$, the position of s_i is preferentially $p(s_i) = pe_k \in ADJ$ to preserve graph locality. The placement concurrently traverses an edge $s_i \rightarrow d_i$ during ITL scanning, and an edge $pe_k \rightarrow pe_j$ in the target graph (lines 5-8). If all $pe_k \in ADJ$ are occupied, the placement searches in the adjacent list of each $pe \in ADJ$, and so on (lines 9-10). The complexity is $O(|ITL||PE|)$ since for each edge in ITL, all PEs will be visited to find an empty place at the worst case. The average case is $O(|ITL||ADJ|)$ as an empty pe is found in the ADJ set during the traversal. Our experiments, in Section 4.5, show that the average measured complexity is $O(3.2|ITL|)$ for 1-hop interconnection.

For the sake of understanding, we provide following examples. Assume we have the dataflow graph from Figure 4.2(a), and a depth-first ITL as shown in Figure 4.6(a). Suppose an initial position for the output node d. The first edge is $c \rightarrow d$, and the ADJ set of d has three pe options as highlighted in Figure 4.6(a). Suppose the east pe is the first option, and the algorithm places c there. The input and target graphs concurrently perform one traversal step, preserving the locality. For any edge $s_i \rightarrow d_i$, the algorithm determines the position of s_i based on the position of d_i, since d_i is already placed in the target architecture, preferentially in the adjacency list. The next

edge is $b \rightarrow c$ as shown in Figure 4.6(b). The ADJ set of c has only two *pe*s, and one is already occupied by d. Therefore, there is only one option to place b below c at the south *pe*. Assume the node a is already visited and placed, as shown Figure 4.6(c). The next edge is $f \rightarrow d$, while d is the destination node, f appears once since all source edge nodes $s_i \in L$ are distinct. Although the ADJ set of d has three nodes, only one is free, and the algorithm places f in the west *pe*. Next, the last edge $e \rightarrow f$ is visited and placed as depicted in Figure 4.6(d).

Routing of the remaining edges

Since ITL is a spanning tree, it only contains a subset of all edges in E for a dataflow graph. Therefore, we should route the remaining edges ($E - ITL$) which are not in the list ITL. For the previous example, $E - ITL = \{(e \rightarrow a)\}$, and the routing finds an optimal routing as depicted in Figure 4.6(d).

Solution Space Exploration

As already mentioned, this algorithm is a fast, greedy approach where the quality of the solution depends on the ITL (line 0), the initial position of placement (line 2), and the target traversal adjacent list (line 4). We propose to execute many instances of this algorithm. For each instance, the initial position, the ADJ list, and the ITL are distinct to explore the P&R solution space.

Figure 4.6(e) shows another example of an instance where the initial position d is at the top-right corner, and the target traversal uses a different sequence in the ITL. This instance also finds an optimal solution. Figure 4.6(f) shows another sample instance found by changing the target graph traversal list (i.e., ADJ list), where a non-optimal solution is found as the edge $e \rightarrow a$ has a non-adjacent routing path. It is essential to highlight that this edge is not in ITL, it belongs to the remaining list, and therefore, this edge is not taken into account during the placement for all instances. However, the greedy strategy reaches the optimal solution for two of three instances in this example.

We summarize our approach as follows: First, we generate an ITL. The next step removes the first edge $s_0 \rightarrow d_0$ and assigns an initial position for d_0. Then, the concurrent traversal in the input sequence and the target graph starts by trying to place node s_i in a free *pe* that belongs to the ADJ set of pe_{d_i}. If there is no free adjacent node, then the algorithm performs a breadth-first search around pe_{d_i} until it finds a free *pe*. The traversal visits all edges in the ITL and places it. However, we must P&R all remaining edges. Previous works uses similar strategies such as depth-first (Ferreira et al., 2007, 2013c), breadth-first (Nowatzki et al., 2018b) or spanning tree (Lai et al., 2005). Also, for fully pipelined architectures (Cong et al., 2014; Nowatzki et al.,

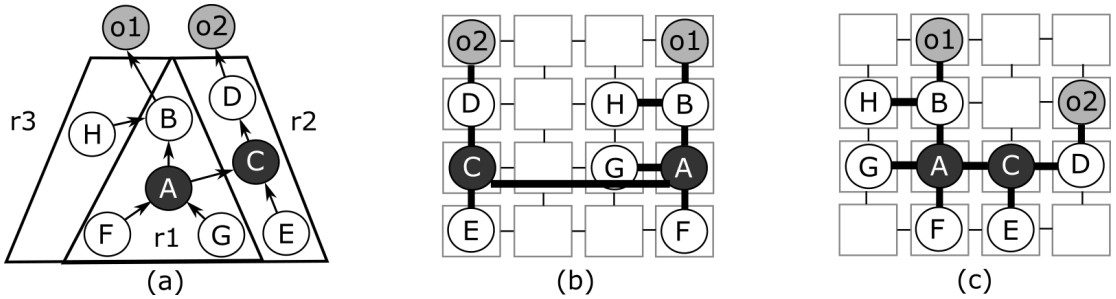

Figura 4.7: (a) Two Outputs; (b) Without correlated outputs; (c) ZigZag traversal.

2018b), the timing step should be yet executed. Our greedy placement is fast and straightforward. However, it can not guarantee that there are feasible routing and timing solutions. We propose to perform multiple P&R instances and then select the best one. In our decoupled approach, the P&R algorithm uses only the wire-length cost function during the single traversal. Once we perform the P&R, we do not allow re-routing an edge nor to change a node position. The final timing step performs a second traversal to get a global view of the P&R solution by computing and optimizing the FIFO sizes.

Our approach proposes to evaluate three parameters to exploit the placement solution space of this greedy traversal heuristic. First, we compare distinct approaches to generate the input traversal list: depth-first (D), breadth-first (B), and our novel zigzag (Z) approach detailed in section 4.3.3. It is important to highlight that there are several distinct lists for any traversal (D, B, or Z). For instance in our example, node d has fan-in two, and the depth-first could start as $c \rightarrow d, b \rightarrow c, \ldots$ or $f \rightarrow d, e \rightarrow f, \ldots$. Second, we investigate how the first node's initial position impacts results. Third, we evaluate distinct adjacent list as detailed in section 4.3.4.

4.3.3 Zigzag Input Traversal

Figure 4.7(a) shows a two output dataflow. For this example, we use a sequence $a \leftarrow b \leftarrow c$ to denote an ITL list $\{b \rightarrow a, c \rightarrow b\}$ with successive edges. Assuming a depth-first traversal starts from o_1 to perform the following sequence: $(o_1 \leftarrow b \leftarrow a \leftarrow f, a \leftarrow g, b \leftarrow h, o_2 \leftarrow d \leftarrow c \leftarrow e)$. All nodes in r_1 and r_2 regions are visited: o_1, b, a, f, g, and h before the output o_2 and its descendants. Therefore, when we place node c, there is no guarantee that it is close to a, since the position of c is fixed to its predecessor d, and the edge $a \rightarrow c$ is not in the ITL. We depict a final placement in Figure 4.7(b) showing nodes a and c far from each other.

The zigzag approach starts as a depth-first traversal from the outputs to the inputs until we find an internal node with a fan out greater than 1. Therefore, the zigzag sequence starts with $o_1 \leftarrow b \leftarrow a$. At this point, we reverse the traversal until we find a

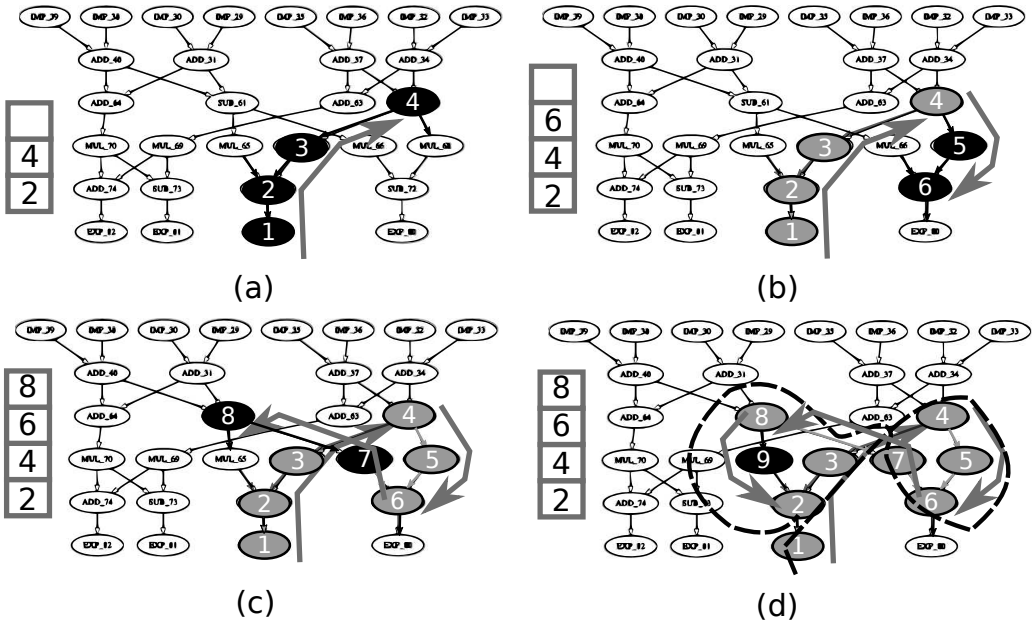

Figura 4.8: ZigZag Traversal: (a) First path; (b) Reverse point and second path; (c) Third path and reverse (node 8); (d) When add the edge $9 \rightarrow 2$, we detect a cycle $(2, 3, \ldots, 8, 9, 2)$, which starts and ends at node 2. Then, we pop the node 8 from the stack to continue the traversal by visiting node 10.

node with a fan in greater than one, a visited node or an output. For our example, we reverse the sequence at a to visit $a \rightarrow c \rightarrow d \rightarrow o_2$. A stack is used to track unvisited edges during the traversal. Figure 4.7(c) shows a possible placement generated from a zigzag traversal where a is close to c, as the position of c is fixed by a.

Figure 4.8 shows a more detailed example of this traversal process. The traversal is bi-directional, and the fan-outs and fan-in higher than one are the reversion points (the idea of zig-zag). During the traversal, if we take the "left path" in a bifurcation, we use a stack to keep track of the "right path" to visit it later. Starting from one output in Figure 4.8(a), the traversal goes in depth-first order until it finds a node with a fan-out higher than one at node 4. We push node 2 into the stack, as it is a bifurcation point, and its fan-in is visited later. We also push node 4 as there are fan-ins to visit later. At the reversion point, we start a depth-first in the opposite direction until finding another reversion point at node 6 (see Figure 4.8(b). At this point, the traversal has captured two correlated outputs. We push node 6 as there are still fan-outs to visit later and reverse the direction. The traversal is reversed again at node 8, as shown in Figure 4.8(c). We traverse until reaching node 2, where we find a cycle. We pop node 8 from the stack to continue the traversal, where node 10 is next to visit, as shown in Figure 4.8(d). At this point, we find the other two correlated outputs. The traversal continues until we visit all nodes. This example demonstrates how we capture multiple correlated outputs.

Finally, the time complexity of our zigzag traversal is equivalent to the time com-

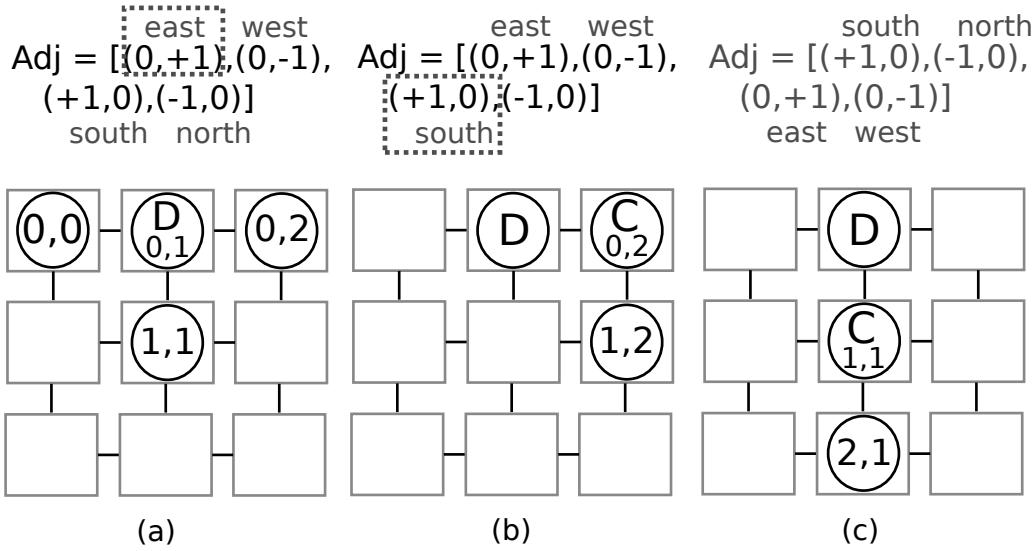

Figura 4.9: (a) An ADJ list as a relative distance vector; (b) Place node *c*; (c) A distinct Adj list generates a new placement of *c*.

plexity of a depth-first traversal, which is $O(|V| + |E|)$, where $|V|$ is the number of vertices, and E is the number of edges.

4.3.4 Adjacency Lists

In addition to the ITL, the adjacency list order (ADJ list) (i.e., the target graph traversal) has an important role in generating the quality of the final placement. For example, the ADJ list has a freedom degree of 4 in a mesh and 8 for the 1-hop interconnected mesh CGRAs. Therefore, the first traversal step has 4 options, and the second step can have up to $4 \cdot 4 = 16$ options, and so on for a mesh. Even though some overlapping options and symmetries reduce this growing number, the solution space multiplies.

The ADJ list represents the target architecture, and we can easily switch from a mesh topology to a 1-hop topology. Also, a simple vector represents the ADJ list for regular structures, as described below. First, we need a matrix to keep track of node placements and free positions. Let us revisit the P&R example from Figure 4.6.

Let us consider the mesh topology where for $pe_{i,j}$: i is the index of the row and j is the index of the column. Figure 4.9(a) shows a simple way to store the ADJ list by using relative index, which represents the directions: east, west, south and north. We create the ADJ list by adding the current *pe* index and the relative index. By following the placement algorithm from Figure 4.5(a), since node d position P_d is (0,1), the first position to be checked for node c is $P_d + Adj[0] = (0,1) + (0,+1) = (0,2)$, which is the east adjacency node as shown in Figure 4.9(a). Then, first free position for node b is $P_c + Adj[0] = (0,2) + (+1,0) = (1,2)$, which is the south adjacent node as

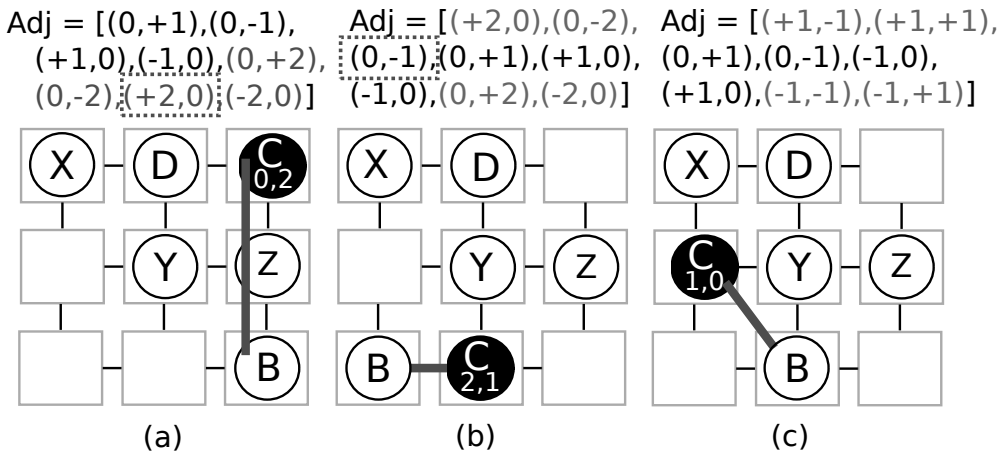

Figura 4.10: (a) 1-hop and placement of b from the edge $b \rightarrow c$; (b) A new ADJ list; (c) Diagonal Interconnections and its ADJ list.

the east is out-of-border and the west is not free as shown in Figure 4.9(b). We can generate a distinct ADJ list by swapping the relative index. There are 4!=24 distinct ADJ lists for a mesh topology. For instance: ADJ_0 = (east,west,south,north), ADJ_1 = (south,north, east, west), and so on. Each greedy placement instance can use a distinct initial position, a distinct ITL (generated by the traversal), and a distinct ADJ list. Figure 4.9(c) shows how a distinct ADJ list generates a distinct target graph traversal and placement.

4.3.5 Interconnection Topologies

Our P&R is architecture-independent, and an ADJ list represents the target architecture. In case of a mesh CGRA, it is straightforward to add 1-hop interconnections to the ADJ lists. Figure 4.10(a) shows an extended ADJ list which includes 1-hop connections. Suppose that we have already placed nodes x, y, and z. When we place node b from the edge $b \rightarrow c$, the only free position is the 1-hop south cell. Figure 4.10(b) depicts a different placement for b as we change the ADJ list. Figure 4.10(c) shows a placement where diagonal connections are added instead of 1-hop, as depicted by the ADJ list. We can add other topologies as Hycube (Walker and Anderson, 2019a), hierarchical clusters, or toroids. Heterogeneous architectures can also be represented as one-memory access per row (see Figure 4.1(d)) from the ADRES-based architectures (Mei et al., 2003; Walker and Anderson, 2019a)).

4.3.6 Modulo Scheduling

The modulo scheduling adds a temporal dimension to the mesh matrix. A common model is to describe the target architecture as a modulo-scheduling routing resource graph (MRRG) (Mei et al., 2003; Hamzeh et al., 2013; Chin and Anderson, 2018a;

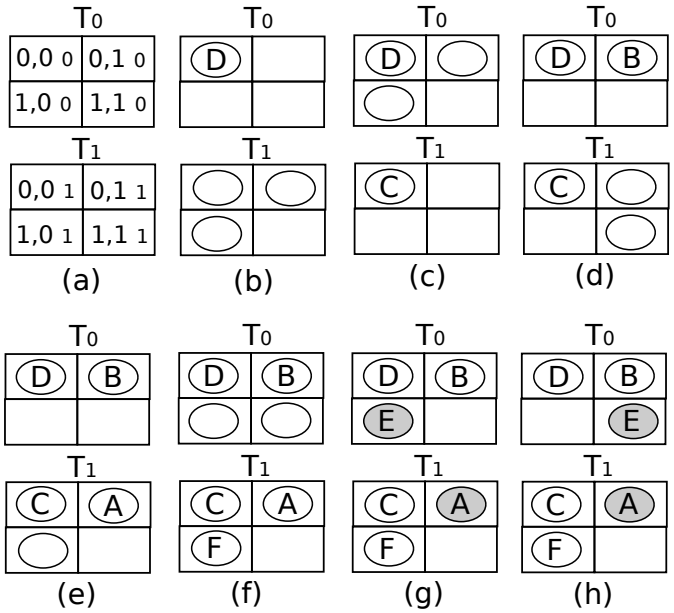

Figura 4.11: Modulo Scheduling: (a) Mesh matrix plus temporal dimension;(b) Initial position; (c-g) Placement sequence; (h) Feasible solution.

Donovick et al., 2019; Walker and Anderson, 2019a), which is a graph that models the hardware connectivity and capabilities included in the temporal partitions. Previous work use MRRG in their approaches: ILP (Chin and Anderson, 2018a; Walker and Anderson, 2019a), simulated annealing (Mei et al., 2003; Hamzeh et al., 2013) and SAT-Solvers (Donovick et al., 2019). Our contribution is that our traversal approach with the MRRG graph includes modulo scheduling in our P&R.

Figure 4.11(a) depicts a simple example for a 2×2 Mesh with initiation interval $II = 2$ (two temporal configurations T_0 and T_1). Each temporal partition stores a CGRA configuration or context (PE functionality and routing). The new ADJ list is {(0,0,**+1**),(0,+1,**+1**),(0,-1,**+1**),(+1,0,**+1**),(0,0,**+1**)} where we add an extra dimension to represent the temporal partition. It's important to note that the node has self-connection (0,0,**+1**).

Considering the same dataflow depicted in Figure 4.3, where node d is placed in $PE_{0,0_0}$ as shown in Figure 4.11(b). There are three possible positions at configuration T_1. Figures 4.11(c-g) shows the following placement sequence: c, b, a, f, and e. However, the first P&R solution violates the latency balancing for the edge $a \rightarrow e$. Figure 4.11(h) depicts a feasible P&R when e uses the second option in the adjacency list.

4.3.7 Elastic CGRA

For the non-optimum feasible solution, we can insert elastic registers (or FIFOs) in the PE interconnections to connect the pipeline stages. The critical question is where these

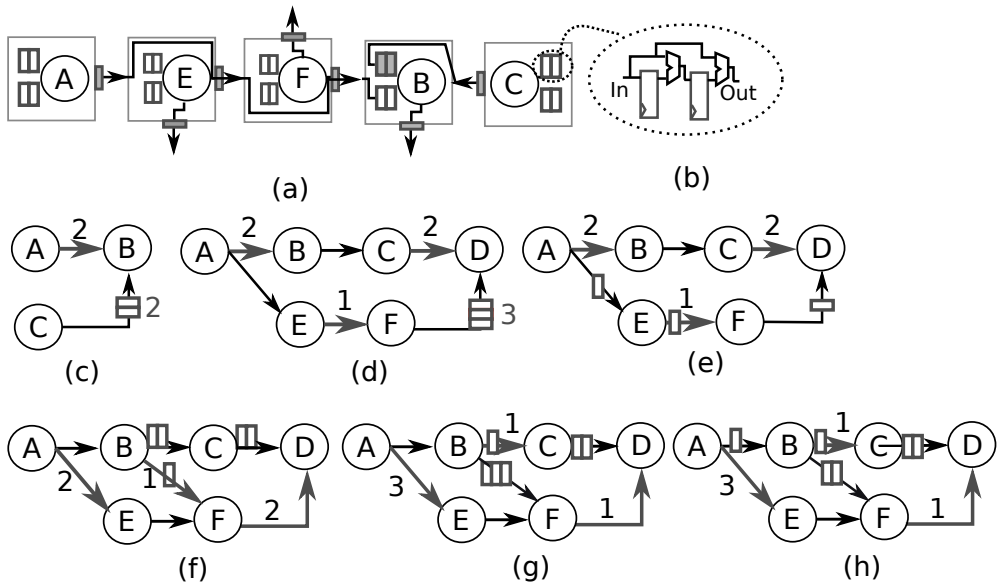

Figura 4.12: (a) Detailed view of elastic CGRA with FIFO size=2; (b) FIFO size implementation; (c) Symbolic view; (d) $LFS = 3$; (e) Splitting the LFS to reduce it to 1; (f) $LFS = 2$; (g) $LFS = 3$ which is reduced to 2 in (h).

FIFO registers should be inserted in the architecture to minimize the total number of registers. For example, a SA approach based on the widespread VPR tool (Luu et al., 2011) is adapted in (Huang et al., 2013) to perform P&R with elastic FIFOs.

We measure the P&R cost as the minimum number of buffering registers required to provide a feasible pipeline implementation for all paths to arrive at the same pipeline stage times. In the presence of reconvergent paths, if one side is unbalanced due to long routing wires, we insert FIFOs on the other side to ensure correct pipeline synchronization.

Figure 4.12(a) depicts a portion of a homogeneous CGRA with FIFO size=2 for each input. It is important to highlight that all PEs have two elastic FIFOs size 2. However, only the PE_b configures one of its FIFO to size 2, and all others PEs bypass the FIFO (i.e., configure as size 0). The edge $a \rightarrow b$ has two extra routing wires that bypass e and f. Therefore, $a \rightarrow b$ has extra wire cost equal to 2. The edge $c \rightarrow b$ has no extra wire. However, it requires a FIFO size 2 to balance the pipeline. Figure 4.12(b) details an elastic FIFO size 2 implemented by multiplexers and registers. For ease of explanation, Figure 4.12(c) shows a symbolic view of Figure 4.12(a) where we insert a FIFO size 2 to balance an extra wire size 2.

It is not straightforward to compute the minimum FIFO size for all connections based only on the extra wiring length. Figure 4.12(d-g) shows examples in order to clarify this retiming/rescheduling problem. Figure 4.12(d) shows a P&R, which has total extra wire length (TEWL) equal to 5, local longest wire (LLW) equal to 2, and longest FIFO size (LFS) equal to 3. Since the path $a \rightarrow b \rightarrow c \rightarrow d$ has extra wire

delay of 4, and the path $a \rightarrow e \rightarrow f \rightarrow d$ of 1, a LFS=3 is inserted between f and d. Nevertheless, the LFS can be split across the path $a \rightarrow e \rightarrow f \rightarrow d$, as shown in Figure 4.12(e), and therefore, a CGRA with maximum LFS=1 can implement it. Figure 4.12(f) shows another example where the TEWL=5 and the LLW=2; however, the LFS is equal to 2. Figure 4.12(g) shows an example where the TEWL=5, the LLW=3, and the LFS is equal to 3. Nevertheless, it is possible to reduce it to 2, as shown in Figure 4.12(h). Finding the optimal LFS value is a hard optimization problem in a decoupled approach (Nowatzki et al., 2018b), where the placement, routing and timing are performed independently. We decide to apply only local transformations to provide a fast timing step, as shown in Figure 4.12, to reach minimum LFS values. First, we compute the arrival time for all edges by traversing the entire mapping graph in topological order from the inputs to the outputs. During the traversal, for each visited node with a delay mismatch, a FIFO is added. Then, we verify if it is possible to split the larger FIFO in smaller FIFOs to be place across the input path. For instance, in Figure 4.12(d), there is a FIFO size 3. This FIFO could be splitted into three FIFO size 1 across the input path $a \rightarrow ... \rightarrow d$ as shown in Figure 4.12(e).

4.4 GPU Graph-based Placement

There are several works describing FPGA placement parallelization, including SA, which partitions the architecture into regions and then parallelize placement for each region (Ababei, 2009) or performs SA swaps in parallel (Wang and Lemieux, 2011; Goeders et al., 2011). GPU placement implementations also include a scalable analytic FPGA placement, as proposed in (Pattison et al., 2015).

In this work, we parallelize our CGRA P&R, performing both latency balancing and modulo scheduling, which we perform on a microsecond scale, thus being suitable for run-time environments. To the best of our knowledge, this is the first work on the acceleration of graph-based CGRA placement on GPUs. Our approach is straightforward and based on parallelizing the graph-based traversals as independent kernel calls. Each GPU thread executes a different instance of the placement kernel, and a post-reduction returns the best solution.

GPU architectures provide highly efficient support for massive parallel thread execution. For example, an Nvidia GTX 1070 has 1,920 cores, which means that the application should launch at least 10,000 threads to hide execution latencies for 5 threads per core. Therefore, if we increase the number of threads from 1,000 to 10,000, the algorithm execution time will not necessarily increase 10x as thousands of threads are hiding the latencies. The question is how to divide our graph traversal attempts along the GPU blocks to exploit local fast memory GPU resources, and at the same time, avoid control divergence execution (Han and Abdelrahman, 2011).

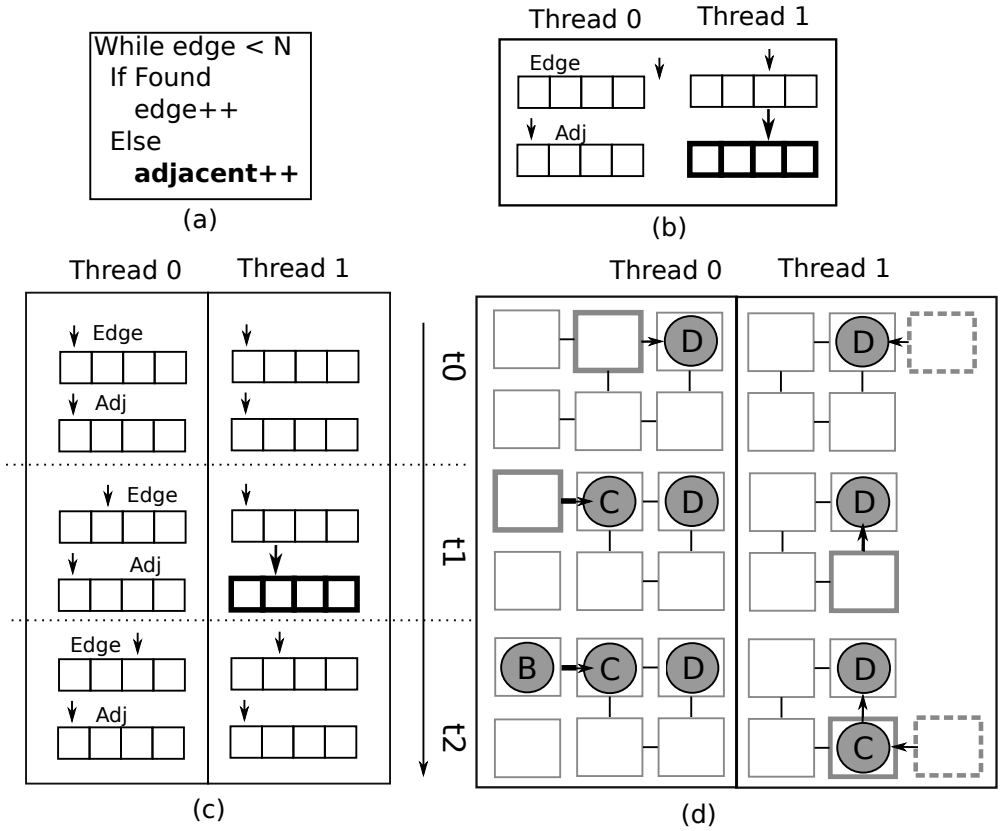

Figura 4.13: (a) Scan list pseudo-code; (b) $Thread_0$ finished first; (c-d) An example of execution sequence for two threads.

Our placement kernel scans two lists: the edge list (ITL) and the adjacency position list (ADJ). The ITL is used to guide the traversal through the graph, while the ADJ is used to find the closest free position on the grid. We mitigate the branch divergence using two pointer indexes, one on each list. If the algorithm finds a free position to place the current node, the main branch increment the ITL index, otherwise, the adjacency position list index is incremented to try the next local position.

During the execution each thread has its own adjacency list. Because of that, one thread can finish early if it is more fortunate to find a free position for an entry in the adjacency list, as shown in Figure 4.13(b). Figure 4.13(c-d) shows an illustrative three-time step example. Threads 0 and 1 initialize and place the starting node D. At t_0, while thread 0 gets the west position to place C successfully, thread 1 fails with a non-existing east position. At t_1, thread 0 also finds a free position to place B, while thread 1 successfully places C at the south position. At t_2, both threads fail.

Our proposed algorithm has a small memory footprint to store the mesh matrices and the node positions. For this data, we use the shared memory, and each warp thread has a reserved bank, where $thread_i$ uses the $bank_{rest\_of\_div(i,32)}$ to avoid conflicts. The multiprocessor of a GPU 1070 has 48 Kbytes of shared memory, and in our experiments, four blocks of 128 threads can execute concurrently. The shared memory

improves the execution by a factor of 3x in comparison to a cache/global memory implementation.

4.5 Experimental Results

In this section, we evaluate and compare our traversal-based approaches. We organize this section follows: Section 4.5.1 introduces the two target architectures, Section 4.5.2 presents the P&R approaches, Section 4.5.3 describes the execution time measures, Section 4.5.4 presents the cost metrics, Section 4.5.5 compares our approach to CGRA-ME (Walker and Anderson, 2019a), and Section 4.5.6 compares to VPR (Luu et al., 2011) and a greedy approach (Ferreira et al., 2007).

4.5.1 Target Architectures

We evaluate two target CGRAs: modulo scheduling and fully pipelined CGRA. For the modulo scheduling experiments, the architecture is an ADRES 4×4 CGRA (Mei et al., 2003) with the initiation interval $II = 2$, 1-hop interconnection network, one memory operation per row is allowed, and with border-located I/O nodes. However, a mesh-based architecture is scalable, and a larger matrix (greater than 4×4) can provide more temporal/spatial parallelism while removing the modulo scheduling complexity by avoiding time-multiplexed contexts.

For the fully pipelined experiments, we evaluate bigger CGRAs with mesh or 1-hop interconnection and elastic FIFOs. We configure this CGRA once before execution, and each PE computes a dataflow node. Therefore, we do not perform instruction fetch/decode at each clock cycle, thus saving power. We consider a homogeneous architecture, where all PEs have the same elastic input FIFO size. We define the mesh size using the minimum square, *i.e.* an N by N matrix where N is the square root (rounded up) of the number of nodes in the benchmark. For example, we map two dataflows with 17 and 62 nodes onto a $5 \times 5 = 25$ and an $8 \times 8 = 64$ PEs architectures, respectively.

4.5.2 P&R Algorithms

We compare our approaches with previous work. The next sections detail the evaluated algorithms and the presentation of the results format.

Previous Work

We compare our work with CGRA-ME (Chin et al., 2017a), VPR (Luu et al., 2011), and a polynomial greedy heuristic based on depth-first traversal (Ferreira et al., 2007).

The open-source CGRA-ME framework supports ILP and SA algorithms. We execute the CGRA-ME with the following parameters: *–II 2 –mapper 0 –cpp 2 –arch-opts cols=4 rows=4*, where *"–II 2"* sets the modulo scheduling with 2 contexts, *"–mapper 0"* selects ILP (*"–mapper 1"* for Simulated Annealing), *"–cpp 2"* determines the C++ architecture to be the Adres CGRA architecture and *"–arch-opts cols=4 rows=4"* determines the gridsize for SA. The CGRA-ME can use several ILP solvers, and we use the academic Gurobi version 9.0 (Gurobi, 2019) as suggested in (Walker and Anderson, 2019a).

The VPR P&R Tool uses the simulated annealing. We measure only the placement execution time in fast mode with the following: *-fast -place_algorithm bounding_box -place_only*. VPR SA is at least 5 orders of magnitude faster than CGRA-ME SA (Chin et al., 2017a). Note, however, VPR's placement does not perform modulo scheduling. Nevertheless, VPR reduces the wire-length, and therefore, the P&R solution can also reduce the FIFO size. We also evaluate a depth-first traversal (labeled as *Depth*, which is a greedy approach that traverses once the graph with a random adjacent vector.

Our approaches

In this work, we present results of single traversal, multiple traversals, branch-and-bound, and multi-thread GPU traversal. For the single traversal, We will label the single traversal *Zigzag*, in which the input list is a Zigzag traversal described in section 4.3.3. We execute this instance with a random initial position and a random adjacent vector. To clarify our notation now, these three elements determine one of our algorithm's input instances: the traversal list, the adjacency list and the initial position. The values for the traversal list and initial positions are randomly set, so that each run potentially generates a different P&R. In addition the traversal list could be one of three kinds: depth-first search (D), breadth-first search (B), Zigzag (Z) or all of them(All). Each instance will be denoted by a tuple (strategy, number of instances). For example, (Z,1000) runs a thousand instances with Zigzag traversal.

The branch-and-bound performs an exhaustive traversal based on a zigzag traversal list, and prune the partial solution if there are no free adjacent nodes. A final solution is not optimal, if all edges in the input list are successfully visited, and one or more remaining edges (not in the traversal list) are not physically adjacent. Because the P&R is an NP-complete problem, we abort this execution when above a threshold time. Finally, the GPU traversal executes 1,024 threads for a zigzag input traversal with a random initial position and a random adjacent vector.

4.5.3 Execution Time Measurement

We measure the execution time using the Chrono library, an Intel(R) Xeon(R) CPU E5-2630v3 2.4 GHZ compiled with GCC 5.4.0 and –O3 option. Our GPU results use an Nvidia 1070 with 1,920 1.5GHz cores.

4.5.4 Metrics

In addition to the execution time, we evaluate the solution cost as a function of required elastic FIFOs. A feasible solution is considered an optimal P&R solution when all adjacent dataflow nodes map to adjacent physical PEs, and there is no need for elastic FIFOs in the target architecture.

Note, as shown in (Nowatzki et al., 2018b), it is hard to solve the P&R without or with small size FIFOs (2 or 3) for mesh topology even for small input graphs (size '20). Moreover, it is not trivial to compute the minimal FIFO size, since it requires a complete graph re-timing, where nested reconvergent paths significantly increase this complexity. For a P&R solution, we apply our greedy heuristic based on the principles shown in Figure 4.12, which is an upper-bound measurement. As future work, we can improve the FIFO distribution to minimize the FIFO size for a given P&R solution.

4.5.5 Results for a Modulo Scheduling CGRA

Table 4.1 shows 15 benchmarks available in the CGRA-ME framework distribution (Chin et al., 2017a), and P&R runtime for various algorithms. Columns Bench and graph size include the benchmark name and the number of nodes for each dataflow graph, respectively. Columns CG_1 and CG_2 show the geometric mean execution time in seconds considering ADRES/Hycube/Cluster architectures as reported in (Walker and Anderson, 2019a). CG_1 uses an ILP approach from (Chin and Anderson, 2018a) and CG_2 uses an improved ILP approach from (Walker and Anderson, 2019a). The three last benchmarks: mults1, 2loop1, and 2loop2, are available in CGRA-ME distribution. However, recent works (Chin and Anderson, 2018a; Walker and Anderson, 2019a) do not evaluate these benchmarks.

Columns ILP CGRA-ME and SA CGRA-ME show the execution time of the ILP and SA approaches obtained by using the CGRA-ME framework (Chin et al., 2017a) with ADRES 4×4 as described in section 4.5.1. Under a time-limit of 1,800 seconds, the ILP approach solves only 12 of 15 benchmarks, and similarly, the SA approach solves 12 of 15 benchmarks (though each P&R fails on a different set).

We report the execution time for a sequential version of our *(All,100)* traversal approach in microseconds. Our greedy heuristic executes up to 100 instances but stops earlier if it reaches a feasible solution. Our algorithm finds a solution for all

Tabela 4.1: Execution Time for a set of benchmarks (Chin et al., 2017a). We compare our traversal proposal with CG_1 (Chin and Anderson, 2018a), CG_2 (Walker and Anderson, 2019a), and CGRA-ME.

Bench	Graph Size	CG_1	CG_2	ILP	SA	Our (All,100)
		CGRA-ME Execution Time in seconds				μs
accum	18	11046.9	20.7	221.6	7201	42.7 μs
cap	24	0.6	0.3	101.4	2763	472.9 μs
conv2	16	3091.3	13.3	122.3	805	9.2 μs
conv3	24	5469.7	31.6	39.1	1080	207.1 μs
mac	11	401.4	5.3	18.3	115	15.9 μs
mac2	24	15372.4	52.7	−−	-	331.8 μs
matrixm	17	1371	10	105.9	1113	139.5 μs
mults2	25	19866.9	52.7	−−	-	501.2 μs
nomem1	6	117.4	2.3	2.6	14	2.7 μs
simple	12	2674.1	7	17.5	555	33.8 μs
simple2	12	2551.5	7.1	17.6	555	24.6 μs
sum	7	82	3.8	5.4	14	3.7 μs
mults1	31	-	-	192.6	4604	670.8 μs
2loops1	11	-	-	19.7	871	12.5 μs
2loops2	16	-	-	−−	-	106.1 μs
geom						54.3 μs

benchmarks, and the geometric mean of our execution time is 54.3μs. Compared to the fastest ILP work (Walker and Anderson, 2019a), which requires a few seconds, our single-threaded algorithm is at least five orders of magnitude faster.

Our placement kernel uses a greedy approach that visits each node once during each traversal in linear complexity. Each P&R instance is different. For the benchmark *simple*, a single execution always finds the optimal solution. For 8 of 15 benchmarks, on average less than 10 greedy execution instances find the optimal solution. Finally, to cover all 15 benchmarks from Table 4.1, the worst-case requires 100 instances. Even for this case, the total execution time is always less than 700 μs.

4.5.6 Results for a Fully Pipelined CGRA

As mentioned in section 4.2.1, a mesh-based architecture is scalable, and in this section, we evaluate our algorithm on a fully-pipeline execution model (Nowatzki et al., 2018b).

In general, the benchmark dataflow kernel sizes for this experiment range from 10 to 100 nodes (Park et al., 2009a). Table 4.2 shows 19 multimedia and digital signal processing benchmarks ranging from 11 to 108 nodes. These benchmarks consist

of a set of representative graph patterns presented in mediabench dataflow graphs. Moreover, we used this benchmark set because it is publicly available on the UCSB website (University of California, 2020), allowing our results to be verified/reproduced, and because it has a diverse graph pattern to evaluate placement and routing strategies. In the following results, we compare our strategies: branch-and-bound, *Zigzag* , *(Z,1000)*, and *GPU* described in section 4.5.2. As a comparison reference, we use VPR (Luu et al., 2011) and *Depth* (Ferreira et al., 2007). We compare execution time and the minimum number of elastic FIFOs as measures of speed and quality of placement. Note that we can emulate CGRA placement with VPR, although this tool targets FPGA technology. All execution time results are for VPR's placement only.

Tabela 4.2: Speed-up evaluation of our Fully pipelined P&R.

			Branch and Bound		Speedup compared to VPR P&R tool (Luu et al., 2011)		
n	Bench	size	Fifo size	Time ms	Zig Zag	Z 1000	GPU
1	mac	11	0	0.05	684.6	350.0	15.7
2	simple	14	0	0.09	587.5	1.5	13.8
3	hornerbs	17	0	0.05	309.7	650.0	8.2
4	conv3	28	0	0.18	542.6	1.7	9.5
5	arf	28	0	0.24s	2110.6	3.4	17.9
6	mults1	23	0	0.1	368.4	1.2	10.5
7	motionvect	32	0	1.4	554.9	22.1	9.1
8	ewf	66	1	>1m	1645.9	3.6	10.6
9	fir2	40	0	0.05	462.4	1.0	5.8
10	fir1	44	0	6.86s	557.3	1.4	5.6
11	Cplx8	77	1	>1m	2611.5	5.6	18.8
12	Fir16	77	1	>1m	1078.4	2.7	7.0
13	h2v2_smo	62	4	>1m	179.3	3.3	17.5
14	feedback	54	0	0.3	711.0	5.7	11.5
15	FilterRGB	84	3	>1m	1094.7	2.0	8.1
16	k4n4op	62	-	>1m	147.1	3.3	17.0
17	cosine1	66	-	>1m	111.2	1.6	6.6
18	cosine2	82	-	>1m	158.7	2.0	7.5
19	interpol	108	-	>1m	163.7	2.6	6.5
Geo					502.1	4.6	10.1

For the exhaustive branch-and-bound approach, we limit the maximum execution time to one minute. Considering 1-hop interconnection, we can find the optimal solution for 10 of the 19 benchmarks, as shown in Table 4.2. For five benchmarks, we find a feasible solution in less than one minute, but these have FIFO sizes greater than 0. Finally, we were not able to find a solution for four benchmarks in less than one

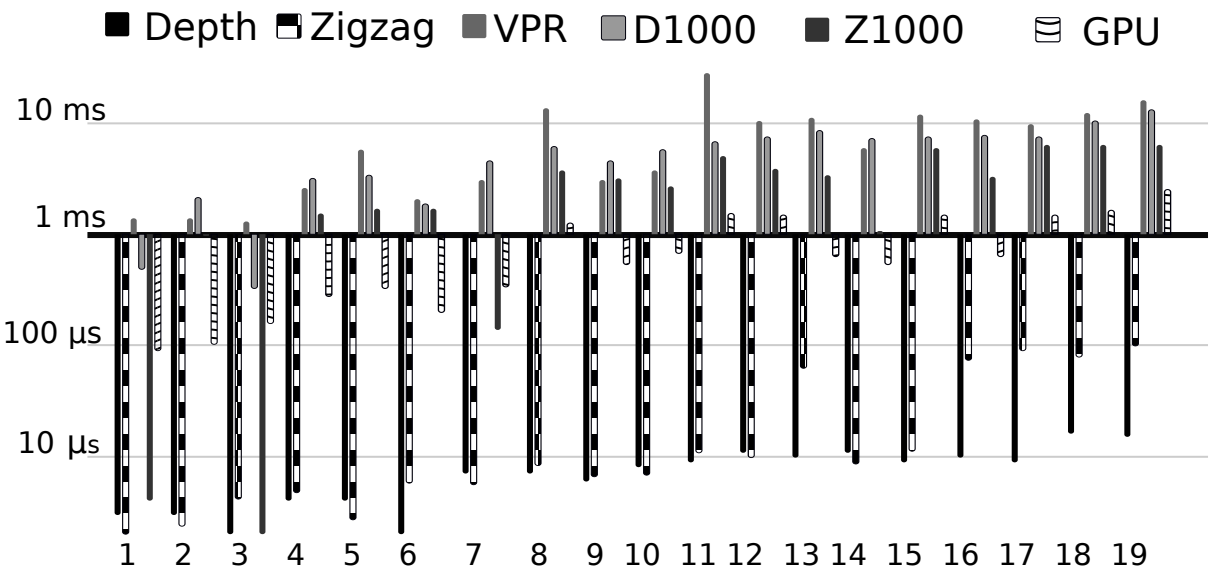

Figura 4.14: Execution for six strategies in 1-hop. Lower is better (log scale).

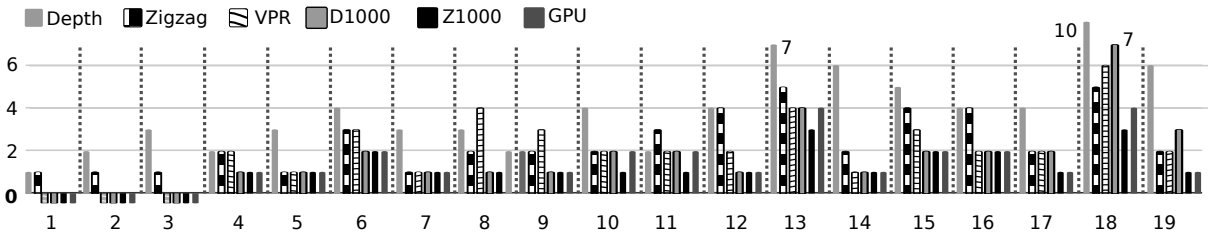

Figura 4.15: Fifo size for six strategies in 1-hop. Lower is better and zero is optimal.

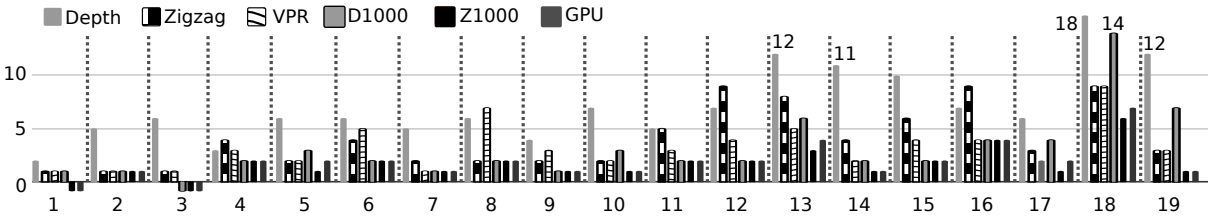

Figura 4.16: Fifo size for six strategies in Mesh topology. Lower is better and zero is optimal.

minute.

Our heuristic approaches do not always find an optimal solution, but they always achieve a feasible solution faster than existing algorithms. We also extend the *Depth* to execute with 1,000 random instances, labeled *(D,1000)*. Figure 4.14 and 4.15 show the execution time and the minimal elastic FIFO size for 1-hop topology and Figure 4.16 shows the elastic FIFO size for mesh topology. We label the benchmarks by the number from Table 4.2. Figure 4.14 shows a bar line graph in log scale with the middle line representing 1ms. The **Depth** (Ferreira et al., 2007) and our **Zigzag** approaches are both orders of magnitude faster (at microseconds scale) since they execute a single traversal. Our **Zigzag** traversal reduces the FIFO size in comparison to **Depth** as

shown in Figure 4.15. The worst-case FIFO size is 5 for zigzag and 10 for depth-first. For medium size graphs ($>$ 40 nodes, indexes 8 to 19), the P&R requires a larger FIFO size.

The last three columns in Table 4.2 shows the speedup in execution time in comparison to VPR, for three of our strategies: single Zig Zag, *(Z,1000)*, and GPU. For 1000 instances, the zigzag ((Z,1000)) is faster than the depth-first traversal ((D,1000)). Also, it reduces the worst-case FIFO size from 7 to 3. Moreover, since we abort execution when we find an optimal solution, for the benchmarks 1, 3, and 7, the (Z,1000) execution time is similar to the single traversal approach.

Although the GPU is the fastest, the (Z,1000) performs a more random search in the solution space, which results in smaller FIFO sizes for 4 benchmarks. In all cases, our traversal approach performs very well compared to VPR SA for these benchmarks on the CGRA, improving the execution time and reducing the average and the worst case FIFO size. The geometric mean of our speedup in comparison to VPR is 502×, 4.6×, and 10.1× for one traversal ZigZag, (Z,1000), and GPU approaches respectively, (see the last row in Table 4.2).

Tabela 4.3: Six Strategies Worst Case and Geometric Means for the FIFO size in a Mesh and an 1-hop topology

	Worst Case					
	Depth	Zigzag	VPR	(D,1000)	(Z,1000)	GPU
mesh	18	9	9	14	6	7
1hop	10	5	6	7	3	4
	Geometric Means					
mesh	6.4	3.2	2.7	2.8	1.7	1.9
1hop	3.5	2.1	2.2	1.7	1.3	1.5

We evaluate FIFO size for a 1-hop and mesh topologies. Figures 4.15 and 4.16 detail the results for each pair strategies/benchmark. We summarize these results in Table 4.3. Our ZigZag improves over the depth-first (Ferreira et al., 2007), and SA VPR approaches (Luu et al., 2011). Moreover, 1-hop also decreases the FIFO size. Previous work, based on a hybrid approach, uses ILP and random breadth-first (Nowatzki et al., 2018b), arguing that it is very hard to solve this problem with FIFO size=3 for the mesh. Although our worst case is 6 for (Z,1000) strategy, 90% of the benchmarks are mapped with FIFO size 3 or less in a few milliseconds compared to dozen of seconds or minutes spent by other approaches (Nowatzki et al., 2018b).

4.6 Conclusion

In this work, we presented a new P&R algorithm kernel for CGRA based on graph traversals, and we explored how to parallelize this kernel on a GPU. We compared our P&R to state-of-the-art SA and ILP P&R algorithms and showed that our algorithm is orders of magnitude faster. We also compared our P&R to VPR's SA placer on a different CGRAs to show the flexibility of our approach: modulo scheduling or fully pipelined, 1-hop or mesh topology, and heuristics for an exhaustive branch-and-bound. Our algorithms produce results that need less FIFO (elastic buffers) in less execution time. Moreover, we introduce a new ZigZag graph traversal that improves previous work based on depth-first (Ferreira et al., 2007), spanning tree (Lai et al., 2005), and breadth-first (Nowatzki et al., 2018b) traversal approaches. The ZigZag traversal effectively captures the internal signals that share outputs. Also, our traversal technique can be used in an exhaustive branch-and-bound approach and can find an optimal solution for graphs with less than 50 nodes.

Due to the lower number of nodes in dataflow graphs mapped to CGRAs, it is possible (as this work demonstrates) that more straightforward graph-based heuristics might be a better choice for solving the P&R problem in this domain. Nevertheless, even for small graphs, the problem is still hard to solve due to the balanced paths (Nowatzki et al., 2018b; Taras and Anderson, 2019). Finally, we have shown how to parallelize these graph-based approaches easily.

In future work, a key question to answer in this space is what complexity of computation graph (defined by the complexity of connections and number of nodes) determines when traditional heuristic placement algorithms such as SA and ILP should be used versus graph traversal algorithms as proposed in this work.

Acknowledgment

We gratefully acknowledge the financial support of the Deutsche Forschungsgemeinschaft (DFG 439918011). This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior–Brasil (CAPES)– Finance Code 001. We want to thank FAPEMIG (APQ-01203-18), CNPq and Intel Academic Compute Environment for financial support.

Capítulo 5

You Only Look Twice: A YOLT Placement and Routing Approach for CGRAs

5.1 Introduction

Coarse-Grained Reconfigurable Arrays (CGRAs) (Podobas et al., 2020) are promising hardware accelerators that architecturally lie between systolic arrays and FPGAs. Systolic arrays provide specialization that results in performance and power efficiency; however, they lack flexibility. CGRAs and FPGAs provide adaptability, especially for emergent demands such as machine learning (ML), where new ML designs and models are exponentially growing in size (Tan et al., 2020). The FPGA bit-level reconfigurability offers flexibility but reduces the power/area efficiency. Furthermore, placement and routing (P&R) requires hours, even days. CGRAs mitigate these issues by using word-level reconfigurability.

FPGA placement and routing (P&R) is a well-known NP-complete problem that still has the potential to be improved (Cong et al., 2003). Although CGRAs implement small design graphs compared to FPGA circuits, the CGRA mapping involves an additional and complex step, the timing or scheduling (Nowatzki et al., 2018b). Furthermore, the timing constraints for CGRA P&R requires that the difference between longest and the shortest paths be minimized (Pasandi and Pedram, 2019). Moreover, it may be necessary to add registers or FIFO registers on routing paths to achieve full path balancing. The complexity of FIFO size is hard when trying to keep FIFO size to 2 or 3 (Nowatzki et al., 2018b), but the problem becomes easier for a FIFO length of 10 or more. However, the cost (in terms of area consumption) of these FIFOs can increase by more than 50% (Nowatzki et al., 2018b).

Traditionally, mapping to CGRAs includes three main steps: placement (P), routing (R), and timing (T) or scheduling. Recently, Nowatzki et al. (Nowatzki et al., 2018b) evaluated several mapping strategies: (1) single-step; (2) three independent steps; (3) a single PR step followed by a timing step; (4) placement followed by RT steps; (5) PR step and followed RT step; (6) single mapping PRT step overlapped by an RT step. Nowatzki et al. (Nowatzki et al., 2018b) argues that strategies 2, 3, 4,

and 5 may fail because the timing phase is challenging to solve independently of the others. However, this analysis includes exact approaches for which the main drawback is the lack of scalability, and the evaluated design graphs are smaller than 25 nodes (Nowatzki et al., 2018b). In a different direction, we propose to adopt strategy 2 and speed up each mapping step without degrading the quality of solutions. Furthermore, we evaluate design graphs with up to 357 nodes and perform the three step mapping process in only a few milliseconds.

In our approach, each step aims to select a solution for the subsequent mapping step that maximizes its chances of success. This work focuses on the placement step and explores the input designs dataflow graph properties and places it with a near-optimal solution that then simplifies the following routing and timing steps. For FPGAs and CGRAs, previous work (Chen et al., 2006; Wang et al., 2017) has proposed a wide range of placement algorithms such as simulated annealing (SA), integer linear programming (ILP), graph traversal (GT), and analytic placement. Recent work includes the use of reinforcement learning (Liu et al., 2019a; Mirhoseini et al., 2020; Goldie and Mirhoseini, 2020) and deep learning (Maarouff et al., 2020; Al-hyari et al., 2020; Alhyari et al., 2019; Lin et al., 2020a) to improve these mapping algorithms. There are pros and cons for each proposed algorithm. For example, in the CGRA mapping domain, ILP (Walker and Anderson, 2019a) does not scale, and SA (Luu et al., 2011; Coole and Stitt, 2010) finds high-quality results, although they are an order of magnitude slower than graph traversal aprroaches (Lai et al., 2005; Canesche et al., 2020).

In this work, we propose a graph traversal algorithm with a tunable execution time that improves solution quality by performing a more deterministic search (still at very efficient run-time speeds (Coole and Stitt, 2015)). Our heuristic provides the best of both goals: high-quality placements similar to SA and the execution time of graph traversal approaches. Our first contribution is to split the graph traveral-based placement into two steps - an annotation traversal and a placement traversal. Our heuristics use a YOTT (you only traverse twice) strategy, in contrast to single traversal approaches (Lai et al., 2005; Canesche et al., 2020) which use a YOTO (you only traverse once) approach. For YOTT, the first step provides annotated information for the second step in a cost-efficient manner. The second step performs placement using the annotated information for certain placement decisions. In particular, we introduce placement guidance for: I/O and reconvergence annotations, degree matching, and look-ahead placement.

YOTT makes deterministic, random, and greedy decisions, and thus, each execution of the algorithm with different initial conditions generates a unique placement solution very quickly. By performing multiple runs, YOTT can explore the placement solution space. YOTT-10 (10-instance executions), for example, improves the

placement quality by a factor of 1.96× and reduces the execution time by 83.4× in comparison to simulated annealing VPR (Luu et al., 2011; Murray et al., 2020). In comparison to YOTO placement (Canesche et al., 2020), YOTT exploits graph properties avoiding the random search of YOTO, which leads to higher quality results in less execution time. For example, YOTT-10 finds results with placement quality similar to YOTO-1000 by reducing the execution time 19.4×. Furthermore, an analysis of the quality of edge placement shows that YOTT-100 optimally maps 90.5% of edges, while VPR only maps 84.9% of these edges.

We organize the remainder of this work as follows. Section 5.2 introduces the basics of placement algorithms and details the single graph traversal approaches. Section 5.3 presents YOTT, explaining the first traversal, I/O, and edge annotations. Also, this section details the local freedom degree and look-ahead placement strategies. Section 5.4 shows our analysis of the YOTT algorithm, comparing it against YOTO and SA. Finally, sections 5.5 and 5.6 present the related work and our conclusions.

5.2 Background

5.2.1 Mind the gap

When we execute CGRA placement as an independent step, we can take advantage of efficient FPGA SA implementations such as VPR (Luu et al., 2011; Murray et al., 2020). VPR performs significantly faster than CGRA SA implementations such as CGRA-ME (Chin and Anderson, 2018b; Chin et al., 2017a), which requires minutes to hours to map a graph with less than 20 nodes. CGRA-ME (Chin and Anderson, 2018b; Chin et al., 2017a) does not decouple placement, routing, and timing steps. Using SA VPR (Luu et al., 2011; Murray et al., 2020) to perform the placement independently, the execution time drops to a few milliseconds/seconds.

Traversal placement approaches (Lai et al., 2005; Canesche et al., 2020) achieves the execution time lower bound by executing just one placement operation per node. Figure 5.1(a) shows the execution time for 23 dataflow graphs, with node size ranging from 11 to 357 nodes. The graphs are ordered by their size and labeled by numbers. We present more details regarding these graphs in section 5.4. For SA approaches, we evaluate two SA VPR (Murray et al., 2020) options: *fast* and *bounding box*, and we also implement SA to perform one execution and 100 executions (SA100), where each one start from a different initial random permutation. The SA execution time is directly related to the number of random swaps, as shown in Figure 5.1. Figure 5.1(a) also depicts the execution time for YOTO traversal approach (Canesche et al., 2020) considering one and 1,000 executions (YOTO-1000). Considering that a single traversal performs one evaluation per node, YOTO-1000 performs 1,000 N evaluations where

N is the number of graph nodes, as shown in Figure 5.1(b). While YOTO is around three orders of magnitude faster than SA approaches, YOTO-1000 execution time is similar to SA approaches and the number of evaluated nodes during the placement.

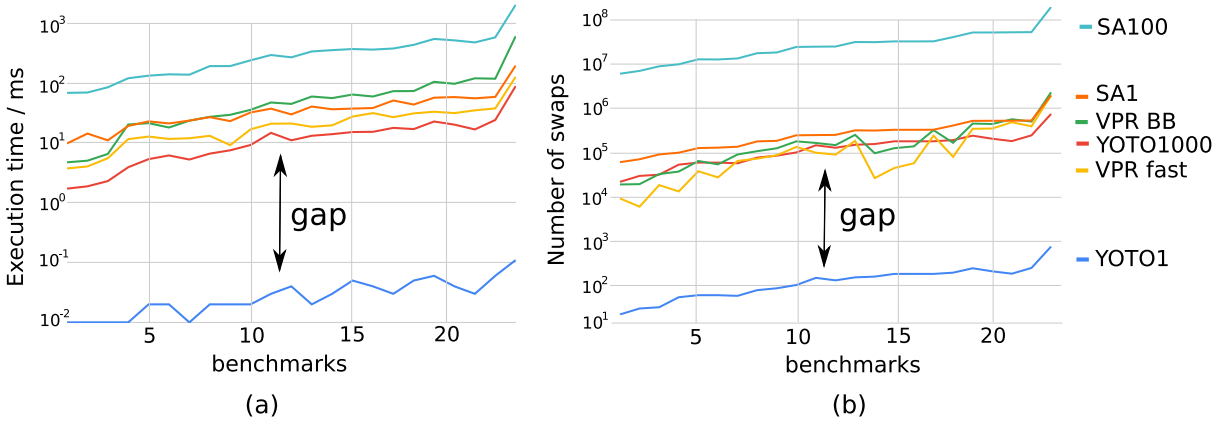

Figura 5.1: (a) Execution time for SA and traversal placements in log scale; (b) Number of swaps for SA approaches and number of evaluation placement for traversal approaches. Lower is better.

We can immediately see that there is a gap in the execution time between the single traversal and SA approaches. Therefore, to exploit this gap to improve the results, we should measure the placement quality before executing the following steps, routing and timing, and after doing so. The SA goal is to minimize the total wire length, and the YOTO goal is to place a node as close as possible to its predecessor during the traversal. One metric to evaluate the impact in the next steps is to measure the number of optimal edge mappings where optimal mapping of an edge $a \rightarrow b$ occurs when the algorithm places a and b in neighbor cells, thus not requiring any routing effort. If the placement of a and b results in non-adjacent cells, the routing step should create a path to connect a to b through a sequence of intermediate cells so that a reaches b.

Figure 5.2(a-c) shows the percentage of optimal edges considering the SA and YOTO approaches. Figure 5.2(a) shows YOTO, VPR fast and SA100. YOTO performs better than VPR fast and quite close to SA100. However, each non-optimal edge could unbalance the mapped graph, and the timing step should add FIFO to balance the paths again. Figure 5.2(d-f) shows the normalized reduction in the FIFO size required to balance all paths where the YOTO is the baseline, and a high value shows a better result reducing the FIFO sizes. Although YOTO performs better on optimal edges, the VPR fast produces more equalized placements, and the final result is better than YOTO. Moreover, the SA100 is much better than both VPR fast and YOTO, as shown in Figure 5.2(d).

Figure 5.2(b) and Figure 5.2(e) compares VPR fast and bounding box (BB). We observe that VPR BB improves both optimal edges and reduces FIFO sizes. However, SA100 performs twice as well as VPR BB in FIFO size reduction. Figure 5.2(c) and

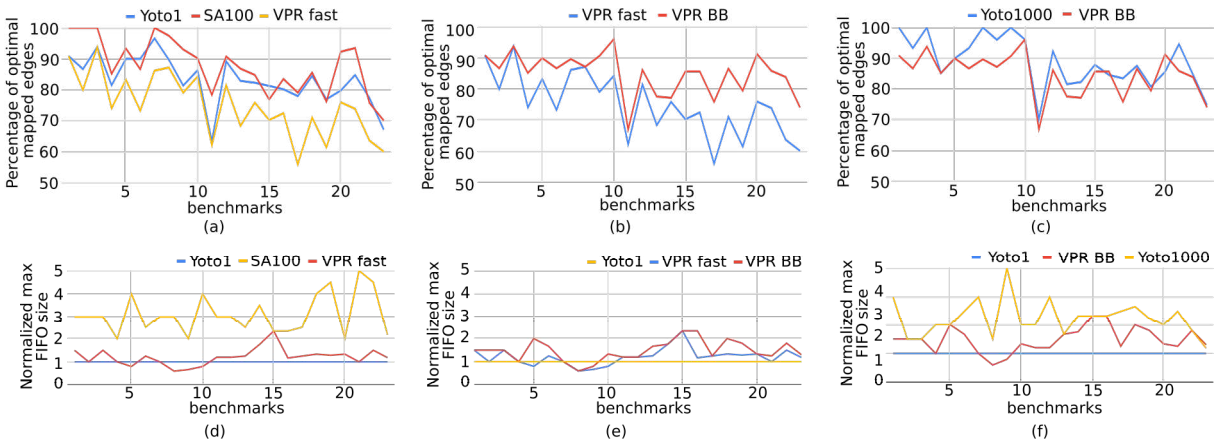

Figura 5.2: (a-c) Percentage of optimal mapped edges for SA and traversal placements, higher is better; (d-f) Normalized maximum FIFO size for SA and traversal approaches, lower is better.

Figure 5.2(e) compares VPR BB and YOTO-1000. Although YOTO-1000 has the execution time and the number of nodes evaluation close to VPR BB as shown in Figure 5.1, YOTO-1000 improves both optimal edges and the FIFO reduction in comparison to VPR BB.

This work shows that there is still room to improve YOTO to get results that are even more comparable to SA100 in terms of quality without a significant increase in execution time. We propose a novel approach by better exploiting the traversal search space. The following sections present and detail the strengths and weaknesses of the previous YOTO approach and how to make further improvements.

5.2.2 YOTO: You Only Traversal Once

Graph mapping translates an input graph to an output graph for the target architecture using a logical node a from the input graph and binding it to a physical cell C_a in the output graph. The placement goal is to optimally map a logical edge $a \leftarrow b$ onto two neighboring cells C_a and C_b. Previous placement approaches (Lai et al., 2005; Canesche et al., 2020) do a single graph traversal for the input and output graphs. Each input graph node is visited once, and we name these approaches as YOTO (You Only Traverse Once). The placement quality depends on both the input graph traversal, which we explain in Section 5.2.3, and on the output graph traversal, detailed in Section 5.2.4.

Figure 5.3 illustrates how a traversal approach makes decisions. First, the traversal should select one starting node in the input graph, as shown in Figure 5.3(a). For this example, the traversal selects node a, noting it could be any node. Then, starting from a, the traversal has two options f and b. Assume the traversal picks node b and then selects node d as shown in Figure 5.3(a). The algorithm traverses both the

input and the output graphs simultaneously and makes decisions on placement in the output graph. Figure 5.3(b) shows the first decision, where it places the first node. Figure 5.3(c) shows after the placement of a, the algorithm must decide where b should be positioned, and Figure 5.3(d-f) shows the options to place d, then c, and finally i.

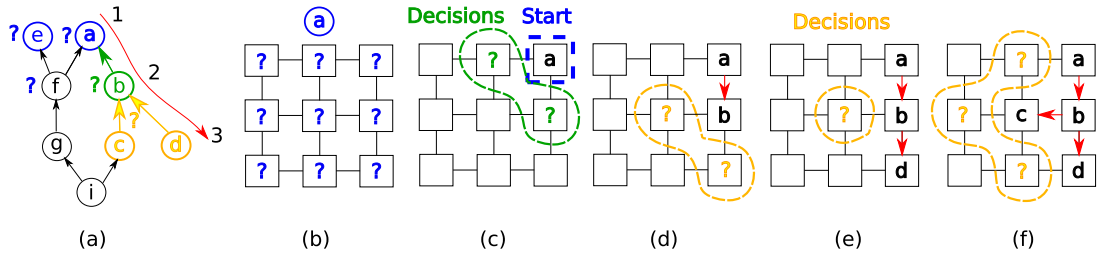

Figura 5.3: (a) Input dataflow graph and traversal decisions; Output Graph: (b) Start node; (c) Start position; (d-f) Neighborhood placement decision.

5.2.3 Input Graph Traversal Order

Traversal order describes how nodes are visited and can be based on existing traversal algorithms such as spanning trees (Lai et al., 2005), depth-first (Ferreira et al., 2007), breadth-first (Nowatzki et al., 2018b), and/or a zig-zag (Canesche et al., 2020). Figure 5.4(a) depicts an input graph and a depth-first traversal, where we use node indexes to depict the traversal order: $a_1, b_2, \ldots$. Figure 5.4(b-g) depicts one possible step-by-step placement for the output graph as a grid. The edge traversal order is $a_1 \leftarrow b_2, b_2 \leftarrow c_3, \ldots, f_6 \leftarrow g_7$. The traversal explores the graph locality by mapping a logical edge to a local physical edge. For this example, the visited edges (black color) result in optimal edges. However, the edges in red color do not belong to the traversal, and the placement decision does not consider them, which results in non-optimal edge mapping, as shown in Figure 5.5(a). The labels on each red edge represent the number of wire segments needed on a mesh topology to route non-adjacent cells.

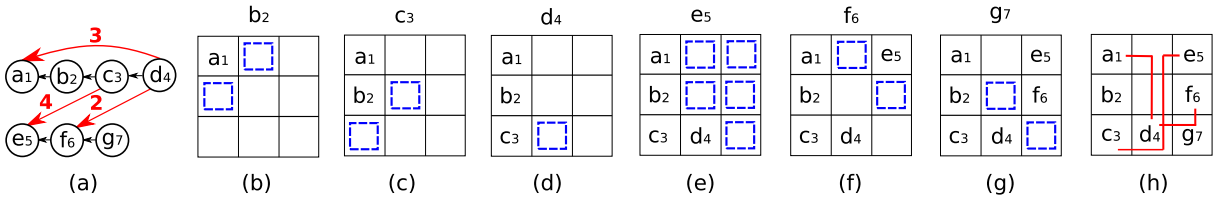

Figura 5.4: (a) An two output data-flow; (b-g) A depth-first traversal placement step by step.

There are two main issues with this approach. First, depth-first based YOTO does not consider unvisited multi-output nodes, which we call correlated output nodes, and these are challenging nodes to place as only one edge path is considered by the

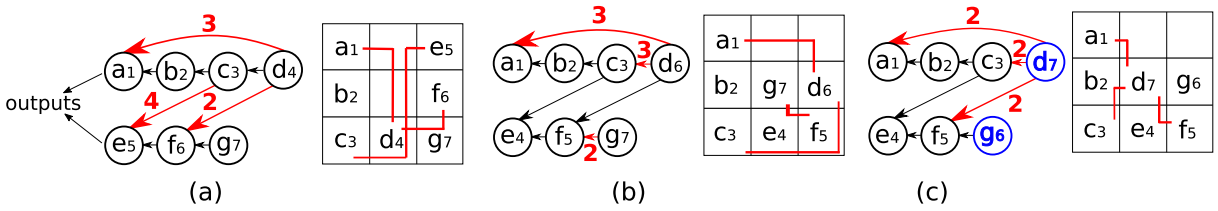

Figura 5.5: (a) Depth-First Order and the final mapping for the example of Figure 5.4; (b) A zig-zag traversal; (c) A new zig-zag traversal.

algorithm when visiting them. In our example, the first visited path starts at output node a and ends at the input node d, and the second path starts at the output e, and no information regarding the position of c and d are considered during the placement of e and f as shown in Figure 5.4(f-g). This results in long wires for edges $e_5 \leftarrow c_3$, and $f_6 \leftarrow d_4$ as shown in Figure 5.5(a). Second, the presence of reconvergent paths can cause problems for YOTO. In our example, only the node c determines the position of d, when the traversal visits edge $c \leftarrow d$. Therefore, node d could be placed far away from a, which leads to a long wire connecting a and d.

One way to mitigate the first issue is by employing a zig-zag traversal (Canesche et al., 2020). Figure 5.5(b) shows a zig-zag traversal, which considers the correlated outputs. When the algorithm finds a fan-out or fan-in greater than one, the traversal is reversed. It starts from the path a, b, and c. Then, the traversal direction is reversed on c, which fan-out (c) > 1. It continues by visiting e, f, d, and g. Nevertheless, there are still two unvisited edges: $c \leftarrow d$, and $a \leftarrow d$, which leads to length three wire segments. Finally, even though the traversal tries to place node g close to f, there are no free slots around f, and $f \leftarrow g$ requires length two wire segments.

A second strategy to make YOTO better is to do multiple mapping instances by randomizing edge selection and choosing the best one (Ferreira et al., 2007), as shown in Figure 5.3. Figure 5.5(c) shows another possible traversal quite similar to the first one, except that edge $f \leftarrow g$ is visited before edge $f \leftarrow d$, which reduces the total routing cost by 2 wire segments.

5.2.4 Output Graph Traversal Order

In addition to different input graph traversals, the output graph can be traversed with different approaches. First, when a node a is placed in a mesh architecture, as shown in Figure 5.6(a), there are at most 4 neighboring possibilities. Assume that we visit edge $a \leftarrow b$. The position of b will be one of the 4 neighbors of a if at least one is available. Previous work (Canesche et al., 2020) approaches this choice by randomly picking a location. Figure 5.6(b) shows a different output traversal considering the same input traversal shown in Figure 5.5(b). It results in a better mapping where only edge $a \leftarrow d$ requires a length 3 wire segment. Figure 5.6(c) shows another output

order, which also results in a near-optimum mapping with a single 3-segment edge for $c \leftarrow d$.

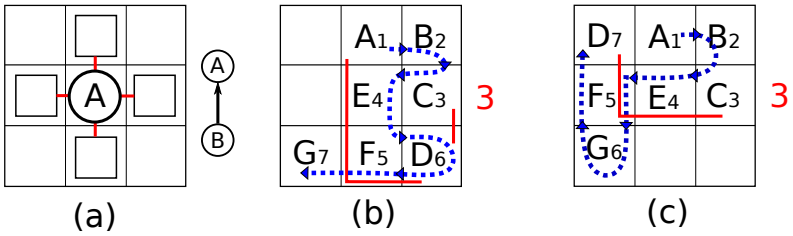

Figura 5.6: (a) Local placement options; (b-c) Two different output traversals (dashed blue line) for the same input traversal from Figure 5.5(b).

Although random input/output orders provide a wide mapping solution space exploration, it can not mitigate reconvergent paths. Moreover, a third issue regarding the input/output (I/O) positions of nodes, which YOTO approaches have not yet constrained can cause poor placements. Typically, I/O connections should be placed at the borders of the CGRA grid. Assume an I/O node has to be placed at the left or bottom border (in those cases, the inputs would be (g and d), and the outputs would be (a and e)). None of the placements shown in Figures 5.5, and 5.6 verify/satisfy this requirement.

5.3 YOTT: You Only Traversal Twice

We propose a novel two-step polynomial placement approach to mitigate the previously described challenges caused by reconvergent paths, I/O requirements, and correlated output nodes. Section 5.3.1 presents the first traversal, which produces the annotations. Section 5.3.2 describes the second input traversal and the placement using annotations, and, finally, Section 5.3.3 presents the pseudo-code of our YOTO approach.

5.3.1 First Input Traversal: Learning and Annotating

The first step of our algorithm traverses all edges in a given order to create the placement sequence and local annotations. For ease of explanation, assume the input graph shown in Figure 5.7(a), performing a depth-first traversal starting from node a. When the node e defines the position of f, it only considers current information, where f should be close to e. It could result in a bad placement, as shown in Figure 5.7(a), where f and d requires a long routing edge. Even by using a zigzag traversal as shown in Figure 5.7(b), node f places the node e without considered the position of the reconvergent node a.

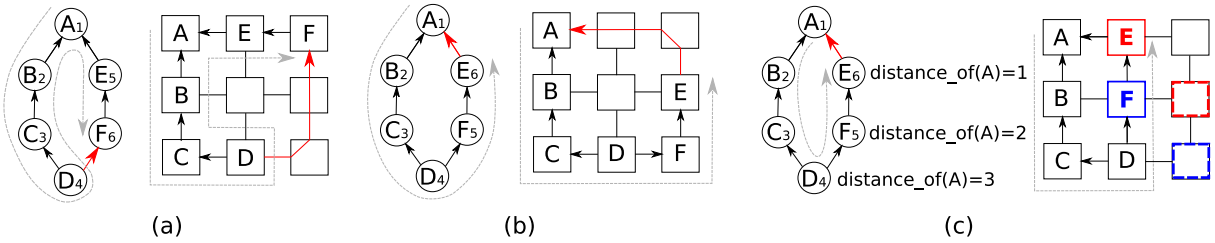

Figura 5.7: Reconvergent example: (a) Depth-first; (b) Zig-zag; (c) Zig-zag plus annotation and optimal mapping.

Reconvergent Annotation

YOTT algorithm will first traverse the input graph to detect reconvergent paths (or cycles) without performing any action on the output graph. Assuming the zig-zag traversal order from Figure 5.7(b): a_1, b_2, c_3, d_4, f_5, and e_6, the YOTT traversal visits all edges. The last edge $e_6 \rightarrow a_1$ detects a cycle or reconvergent path. The YOTT algorithm will annotate at the edge $e \rightarrow a$ that e should be at distance one from a. Next, the YOTT performs a "backpropagation"to the predecessor edge $f \rightarrow e$ to add the annotation of distance 2 of a, and next to the predecessor edge $d \rightarrow f$ to add the annotation of distance 3 of a as shown in Figure 5.8.

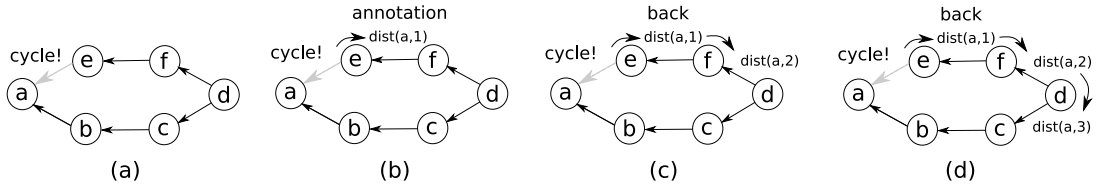

Figura 5.8: First Traversal and annotations: (a) Detect a reconvergent or cycle; (b) Create an annotation; (c-d) backpropagate the annotation to the predecessor edges in reversal traversal order.

After YOTT writes these annotations, the second traversal visits simultaneously both the input and output graphs. If there is no edge annotation, the YOTT second traversal behaves as YOTO first traversal, by randomly picking a neighbor cell. However when there is an annotation, it determines the preferential position. For our example, the annotation distance 2 of a places f at the top of d instead of at the right as shown in bold color in Figure 5.7(c), and finally, the annotation distance 1 of a places e at the top of f, which generates an optimal mapping. YOTT works since a appears first in the traversal order, before e and f, then when these nodes are placed, the physical position of a is known, and e and f placements are guided towards a. Therefore, the first traversal collects the graph reconvergent data to guide the second traversal changing random decisions to deterministic ones.

I/O Annotation

The second annotation guides the placement of I/O nodes and their predecessors. The basic principle is to backpropagate the I/O node distance guideline to predecessor edges. Assuming a simple graph depicted in Figure 5.9(a), where the nodes a and f are I/Os that should be located at the grid borders. Figure 5.9(b) shows a placement that does not consider f I/O information. During the first traversal, YOTT also adds I/O annotations as shown in Figure 5.9(a) by labeling the edges $d \rightarrow e$ and $c \rightarrow d$ since these edges are the predecessors that will impact the placement of node f. YOTT second traversal has three options for d that satisfy the I/O annotations as shown in red color in Figure 5.9(c), where one is picked up. Next, YOTT has two options for e, and it avoids the bottom position, which does not satisfy the I/O annotation distance of 1.

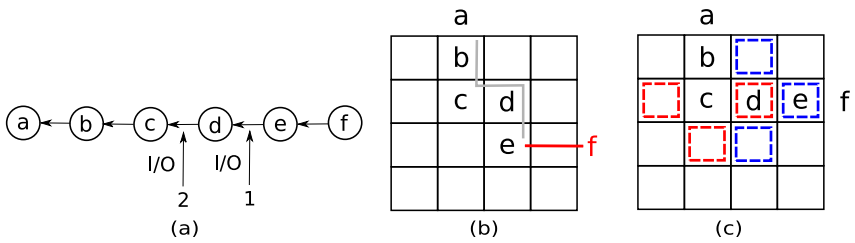

Figura 5.9: (a) Data-flow Graph plus I/O annotation for node f; (b) Placement without annotations; (c) Placement guided by I/O annotations.

5.3.2 Second step: Placement Guidance

For the second step, YOTT places the nodes with two additional contributions called degree matching in section 5.3.2 and Look-ahead in section 5.3.2. Otherwise, this step is the same as YOTO based placement.

Degree Matching

We propose a strategy to improve the random node selection on the output graph based on selecting nodes with higher degrees of freedom (number of unoccupied neighbor nodes) similar to or match the to-be placed logical node. Assume the data-flow graph in Figure 5.10(a), where we have already placed a and b, and we are looking for the best position to place c, that has a degree 3 in the input graph and b also has three options of where to place c, as shown in Figure 5.10(b) for a mesh architecture. Consider also that there are already other occupied nodes in the grid marked by a $\times$. We use the term cell to refer to a node in the output graph (target architecture). Figure 5.10(c) displays the number of free edges for each cell in the output graph for each position. For instance, the number 2 at the right side of b (blue

color) represents two additional free positions (north and south). The number 1 at the bottom of b represents one east free cell and the number of 1 at the top of b. For our example, we can select the left cell to place c since it has enough free cells to place the in-edges d and e for the node c as shown in Figure 5.10(d).

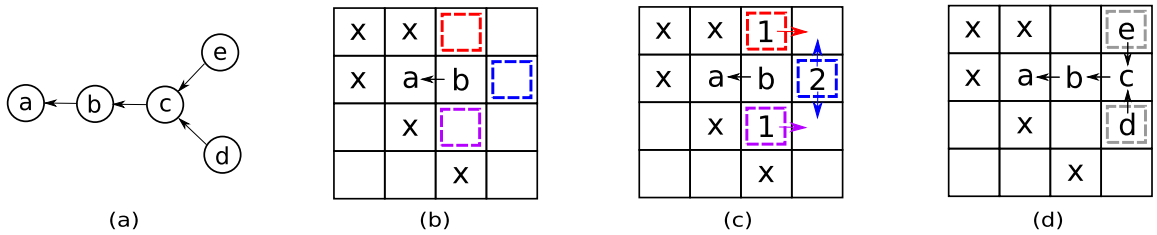

Figura 5.10: Degree Matching: (a) Data-flow example; (b) Mesh after the placement of a and b; (c) Number of free cell if c is placed in the dotted boxes; (d) Best option to place c to satisfy the node degree.

Look-ahead

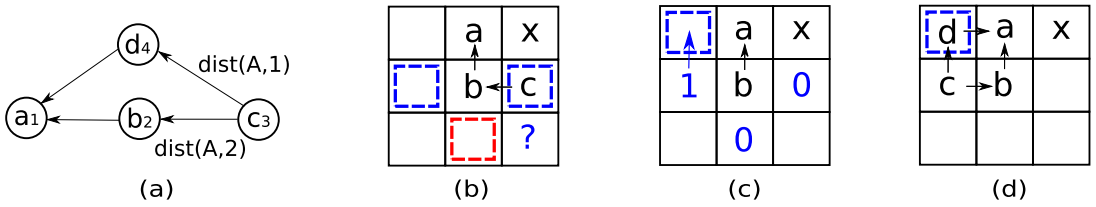

Figura 5.11: (a) Annotated Data-flow. (b) The placement of c without look-ahead; (c) Number of free cells at distance one of a to place c followed by the placement of d; (d) The placement of c guided with look-ahead.

Placement now considers two edges instead of one in the presence of annotations, called look-ahead. We use the look-ahead strategy to process distance-two annotations. Consider the simple graph shown in Figure 5.11(a) for a mesh architecture. We are looking to place c close to b at distance 2 of a (guideline annotation). The bottom cell in red is distance 4 of a. There are two free distance-2 cells (east and west), as highlighted in blue color. However, only one of them will be optimal. Assume that we place c at the bottom of b, as shown in Figure 5.11(b). Then, we process the next edge $d \leftarrow c$, d is placed at the bottom of c, and therefore it fails, d will be at distance 2 of a. We propose to look ahead one step to select a promising distance-2 cell, ensuring the availability of a distance-1 cell. For our previous example, if we choose the left cell to place c, as shown in Figure 5.11(d), there is a free cell at distance 1 of a to place d. The labels inside the blue boxes in Figure 5.11(c) show the number of cells that will be at distance 1 of a in the next step. There is only one option around b.

5.3.3 Algorithm

Figure 5.12 shows the pseudo-code for the first step of YOTT. The complexity of this algorithm is $O(|E|)$, where E is the set of input graph edges (each represented as a pair of nodes). We start by adding the output nodes onto a stack. Lines 2-3 get one output as the first placed node in the edge sequence. Lines 4-15 are the main core of the algorithm, where each iteration visits one edge. This edge has two nodes, one is the *placed_node*. This node has already been placed when it appears in the edge sequence. The second one is the *target_node*, which position should be in the neighborhood of the *placed_node*. For ease of explanation, we use p as the *placed_node* and t as the *target_node*. Each iteration processes the edge $p - t$, where t should be placed next to p.

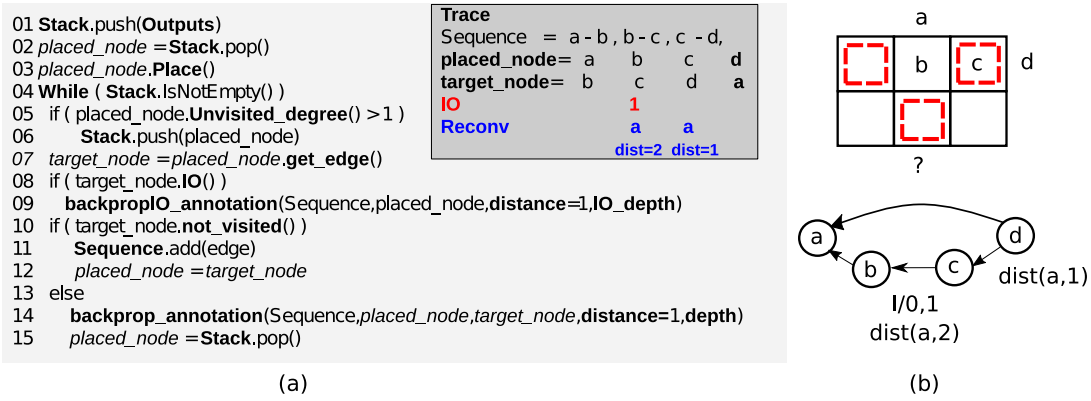

Figura 5.12: Pseudo-code of first traversal and annotation step.

Lines 5-6 verify if p has more than one unvisited edge and push p onto the stack to visit these other edges later. Line 7 defines the traversal order when it picks an unvisited edge $p \leftarrow t$. The traversal algorithm can use strategies such as depth-first (Ferreira et al., 2007), breadth-first (Nowatzki et al., 2018b), or zig-zag (Canesche et al., 2020).

Lines 8-9 check and call the function *backpropIO_annotation* if t is an I/O node. Let us consider the example shown in Figure 5.12(b), for which we show a trace in Figure 5.12(a). When we process edge $c \leftarrow d$, $t = d$ is an I/O node. Therefore, the function *backpropIO_annotation* visits the predecessor sequence edges to add the I/O annotation distance. The initial I/O distance is set to one. The annotation can be back-propagated up to n predecessor edges, where the parameter *IO_depth* defines n. The first step also adds to reconvergent annotations at edges $b \leftarrow c$ and $c \leftarrow d$, that guides the placement of d close to its predecessor c and the annotation a.

Lines 10-12 add the current edge to the sequence, and, therefore, the p neighborhood will define the position of t. Moreover, t becomes a *placed_node* for the next iteration. Finally, lines 13-15 handle the case where t has been already visited, and

there is a reconvergent edge (or a cycle). Function *backprop_annotation* will propagate it to the predecessor edges.

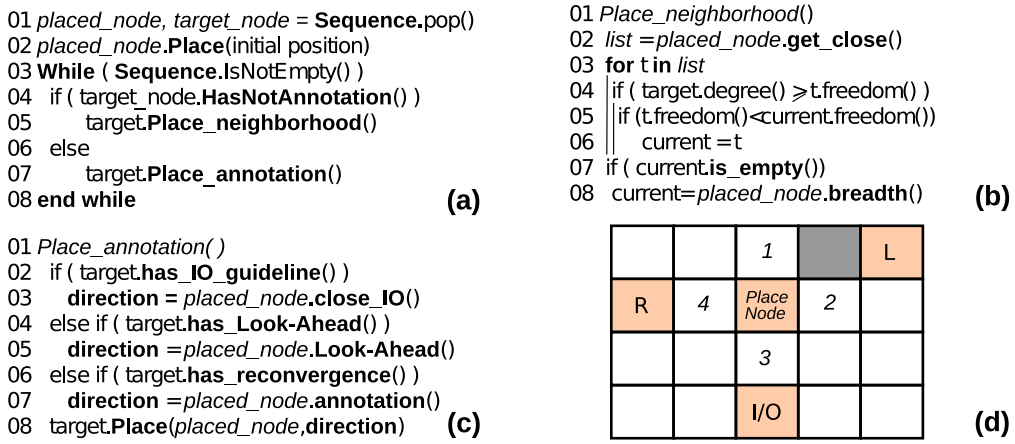

Figura 5.13: Pseudo-code of the second traversal and placement step

Figure 5.13(a) shows the pseudo-code for the second traversal step. We start by scanning the annotated edge sequence. The complexity is approximately $O(|S| \times N)$, where $|S|$ and N represent the sequence size and the number of evaluated neighbors during a placement, respectively.

If there is no annotation, the function *placement_neighborhood*, shown in Figure 5.13(b), is called. Line 2 gets a list of the closest neighbor cells. By default, we assume a list of distance one from the *placed_node* in random order. Lines 3-6 select the free cell with the best match in terms of edge degree for the *target_node*. Finally, in the worst case, when there is no free cell available, we perform a greedy breadth-first search around the *placed_node* and pick the first free cell.

When there are annotations, the YOTT-based approach uses the function *Place_Annotation* shown in Figure 5.13(c). First, we prioritize the I/O distance. Consider *place_node* shown in Figure 5.13(d), which has an I/O guideline of distance one of I, the direction 3 will be selected. The second priority is the look-ahead annotation. Consider a distance two of the target node L shown in Figure 5.13(d). The look-ahead technique selects direction 2 since direction 1 has no free cell at a distance one from L. Finally, if there is a reconvergent path annotation at distance one of R, we place the target node in direction 4.

In summary, we have three annotation classes: I/O, lookahead, and reconvergence. If we have more than one annotation per node, we establish a class-based priority order. The highest priority is to place I/O nodes in positions not too far from the borders, where the PEs that can handle I/Os are located. When our algorithm places nodes that converge to I/Os close to the border, we save routing and FIFO resources. The second priority concerns the lookahead annotation. When we treat

the reconvergence in advance (distance of 2), the algorithm reaches a placement with fewer non-optimal edges, thus saving routing resources.

5.4 Experimental Results

We evaluated our approach by using a representative set of dataflow benchmarks provided by the University of California, Santa Barbara (University of California, 2020). We also included dataflows from the CGRA-ME framework (Chin et al., 2017b). Our benchmarks are available at (Benchmarks, 2020). The target CGRA is a one-hop topology (Bansal et al., 2004a; Mei et al., 2007), where each node gets a dedicated cell, and the data has to arrive at a specific cycle in the fully pipelined mode (Nowatzki et al., 2018b). We always use the minimum possible CGRA size, which is the minimum square grid. Assuming an N dataflow node, the grid size is $\lceil\sqrt{N}\rceil \times \lceil\sqrt{N}\rceil$. For example, when we consider $N = 30$, then the grid is $\lceil\sqrt{30}\rceil \times \lceil\sqrt{30}\rceil = 6 \times 6$. The minimum grid increases the mapping challenges to reach optimal solutions.

We compared our results against a small set of CGRA mapping tools as there are not that many. CGRA-ME (Chin et al., 2017a) and an agile SMT-Based Mapping (Donovick et al., 2019) are available. However, they are an order of magnitude slower than VPR (Murray et al., 2020) and YOTO (Canesche et al., 2020), and they do not scale to graph size larger than 30 nodes, since both implement exact approaches (integer linear programming and Satisfiability Modulo Theories (SMT), respectively). We, therefore, propose to compare the YOTT approach against SA on VPR (Murray et al., 2020) and a recent YOTO traversal approach (Canesche et al., 2020). We also implemented a parallel version of the SA-100 approach in Open-MP to perform 100 executions starting from a different random permutation. The number of nodes in these datasets ranges from 11 to 357.

We run a series of experiments to quantify and analyze these three approaches as follows: Section 5.4.1 presents a summary comparison of the three approaches. Section 5.4.2 provides a wire length analysis. Section 5.4.3 details the benchmarks to provide deeper insights on edge properties and their impact on the algorithms. Section 5.4.4 analyzes the percentage of optimal edge mapping as a function of the graph proprieties and the mapping effectiveness. Section 5.4.5 includes execution time analysis and the node evaluations and swaps number for traversal and SA approaches, respectively. Section 5.4.6 presents mapping quality focusing on pipelining execution by minimizing FIFO sizes. Section 5.4.7 evaluates the impact of reconvergent annotations.

5.4.1 Algorithm Evaluation Summary

Table 5.1 summarizes the placement execution time, the wire length, and FIFO sizes. The wire length metric guides the placement, and it is executed as an independent step. It is a local and straightforward metric that provides quality estimation for further steps without increasing placement execution time. We map the dataflows onto a one-hop, fully pipelined CGRA. Therefore, the throughput is always one result per clock cycle. As presented in (Nowatzki et al., 2018b), elastic size FIFOs fix the delay mismatch paths to ensure correct functionality. However, FIFO size has a significant impact on the final architectures area cost (Nowatzki et al., 2018b), and it is, arguably, the most important metric to optimize.

Experiments have been performed on an Intel(R) Xeon(R) E5-2630v3 2.4 GHz CPU. The implementation has been compiled with GCC 5.4.0 using the optimization flag -O3 and the execution time has been measured using the Chrono library. We measured VPR 8.0 (Murray et al., 2020) placement execution time by using the following parameters: *-place_algorithm bounding_box -place_only*. We also evaluated VPR in fast mode by using *-fast*, and routing time is not included in our measurements. We consider VPR placement time as our baseline. Row *speed* in Table 5.1 shows the speedup ratio relative to VPR. We did not make any modification to VPR. We performed the placement, routing, and timing as separate and independent steps to reduce the placement execution time. We used VPR to perform the placement targeting the minimum wire length cost function. YOTT and YOTO also perform the placement by using only the wire length as a cost function. VPR (Murray et al., 2020) is very efficient for large as well as for small graphs, and to the best of our knowledge, VPR is the best SA implementation to be used as reference point.

Tabela 5.1: Speedup Ratio (in comparison against VPR), Average FIFO Size, Average Wire Length in Segments

	VPR		YOTO	**YOTT**	YOTO	**YOTT**	YOTO	**YOTT**	YOTO	SA
	BB	Fast	1	**1**	10	**10**	100	**100**	1000	100
speed	1	2.41	1848	766	302	83.4	32.6	9.7	4.2	0.16

Average Speed up compare to VPR (high is better)

wires	1.25	1.31	1.30	1.27	1.24	1.18	1.20	1.16	1.17	1.16

Average wire length in segments (low is better)

fifo	1	0.80	0.94	1.66	1.35	1.96	1.49	2.24	1.92	2.47

Ratio of FIFO Size in comparison to VPR bound boxing (BB), high is better

If the main goal is the maximum performance, a single execution of the YOTO-

1 and YOTT-1 reduces the VPR execution time by 1848× and 766×, respectively. We observe that the proposed YOTT is, on average, 2.4× slower than YOTO due to performing two traversals and further propagations. Nevertheless, YOTT reduces the average FIFO size by 1.66× compared to VPR, while YOTO has worse results than VPR.

We also provide YOTO and YOTT with multiple instances to explore better the mapping solution space. Since both approaches take random decisions during the traversals, each execution may result in a different solution cost. YOTO-N and YOTT-N represent the corresponding algorithm running N times (its result is the best one among the N obtained). Execution time increases linearly with N.

If the goal is to reduce the FIFO sizes and the wire length, YOTT-100 reaches the same wire length as SA-100. Furthermore, YOTT reduces the FIFO size by 2.24× and 1.17× in comparison to VPR and YOTO-1000, respectively. Moreover, it is 9.7×, and 2.3× faster, respectively. Finally, YOTT-100 is 60.6× faster than SA-100, and it is only 9% worse in FIFO size reduction. Therefore, YOTT is as fast as a traversal approach, and it produces high-quality mapping similar to simulated annealing.

5.4.2 Looking under the Hood: Average Wire Length

Row *wires* of Table 5.1 presents the average number of wire segments needed to implement an edge in the one-hop target architecture. Although all mappings show quite similar wire length results, there is a significant difference in FIFO reduction and execution time. This section analyzes this issue.

For ease of explanation, we first present an example. Figure 5.14(a) shows a graph with 11 nodes and 12 edges and its YOTO mapping. Ten edges require only one wire segment, as shown in the grid below the graph. However, edge $c \leftarrow f$ requires three wire segments (in red color), and the edge $g \leftarrow k$ requires 6 wire segments, assuming a mesh topology. The average number of wire segment is $\frac{10\times1+1\times3+1\times6}{12} = 1.6$.

We propose to classify the edges into three categories to understand the traversal strongness and weakness better. The YOTO approach uses only two of them, as shown in Figure 5.14(a). The category *first* includes the first time a node is visited, which defines its placement. The sequence starts on $a, c, e, h, \ldots$ labeled by $1, 2, 3, 4, \ldots$, respectively. There are 10 in 12 edges (82%) in category *first*. The YOTO goal is to optimally map these edges by using a single segment. However, due to random decisions taken during the traversal and the lack of routing resources, edge $g \leftarrow k$ requires 6 wires. Therefore, the average wire length for the $first$ category is 1.5.

The second category is the reconvergent edges. For our example, $c \leftarrow f$ and $f \leftarrow j$ (in red color) are classified as reconvergent edges, which have been not visited during the zigzag YOTO traversal shown in Figure 5.14(a). Fortunately, f is placed next to

c, and only the edge $f \leftarrow j$ requires 3 wires. The average wire length for the *reconv* category is 2, since these edges are random or unpredicted for a YOTO placement.

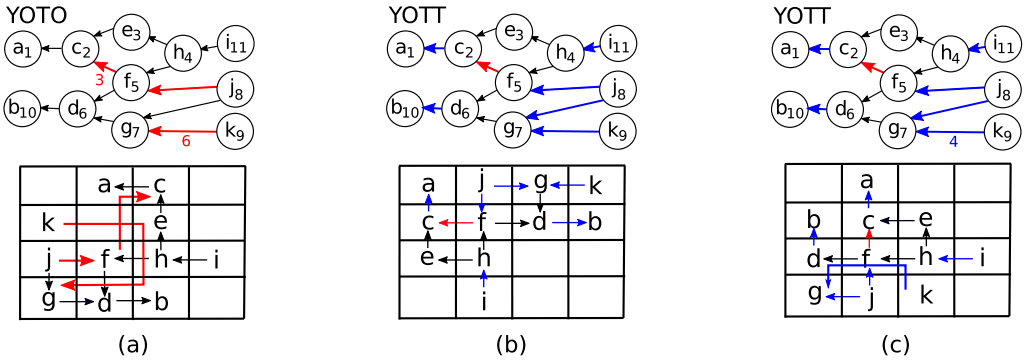

Figura 5.14: Edge distribution categories and a placement:(a) A YOTO instance; (b-c) Two YOTT Instances.

YOTT introduces a third category, *I/O*, which includes any edge that contains an I/O node. Figure 5.14(b-c) shows the same example where the I/O, $first$, and *reconv* edges are depicted in blue, black, and red colors, respectively. YOTT targets to optimize all categories. Considering the same first traversal applied in YOTO, thanks to the annotations and placement decisions, YOTT reaches an optimal solution shown in Figure 5.14(b) and a near-optimal in Figure 5.14(c). Therefore, YOTT requires fewer random decisions during placement.

On the other hand, VPR (Murray et al., 2020) is not based on the edge category. For comparison purposes, we propose to classify all VPR edges. We label the edges that include an I/O as an I/O edge. For the nodes with a fan-out of one, we mark the respective output edge as $first$. In fan-outs greater than one, the edge that requires fewer wire segments is classified as $first$, and the remaining edges are included in the *reconv* category.

Tabela 5.2: Three Edge Categories: First, Reconvergent an I/Os.

	VPR		YOTO	**YOTT**	YOTO	**YOTT**	YOTO	**YOTT**	YOTO	SA
	BB	Fast	1	**1**	10	**10**	100	**100**	1000	100
Total	1.25	1.31	1.30	1.27	1.24	1.18	1.20	1.16	1.17	1.16
first	1.16	1.27	1.15	1.10	1.07	1.05	1.06	1.03	1.05	1.03
reconv	2.54	2.52	2.97	2.56	2.58	2.20	2.44	1.98	2.32	2.13
IO	1.20	1.17	-	1.24	-	1.20	-	1.18	-	1.23

In terms of these three classifications, Table 5.2 shows that YOLT-100 reduces the average wire length in all categories. Therefore, the YOTT approach uses annotations to exploit the graph properties better and enhance mapping quality. Another interesting result of YOTT-10 is that the average I/O edges are almost the same as VPR,

showing the effectiveness of I/O annotations. The main challenge in getting a good placement is the reconvergent edges. The annotation and look-ahead mitigate it, but there is still room for improvements.

5.4.3 Edge Distribution

Tabela 5.3: A set of dataflow graph benchmarks provided by UCSB (University of California, 2020) and CGRA-ME framework (Chin et al., 2017a)

			YOLT / YOLO Percentage of total edges		
n	bench	I/O:V:E	IO	First	Recon
1	mac	3/1, 11, 11	36 / -	55 / 91	9 / 9
2	simple	4/1, 14, 15	40 / -	47 / 87	13 / 13
3	horner_bs	4/1, 17, 16	37 / -	63 / 100	0 / 0
4	mults1	1/1, 24, 27	18 / -	78 / 85	4 / 15
5	arf	8/2, 28, 30	40 / -	47 / 87	13 / 13
6	conv3	9/1, 28, 30	37 / -	53 / 90	10 / 10
7	motion_vec	14/3, 32, 29	65 / -	35 / 100	0 / 0
8	fir2	16/1, 40, 39	44 / -	56 / 100	0 / 0
9	fir1	22/1, 44, 43	53 / -	47 / 100	0 / 0
10	fdback_pts	21/5, 54, 51	61 / -	35 / 96	4 / 4
11	k4n4op	22/2, 59, 74	51 / -	43 / 77	5 / 23
12	h2v2_smo	16/1, 62, 65	26 / -	68 / 94	6 / 6
13	cosine1	16/8, 66, 76	31 / -	45 / 76	24 / 24
14	ewf	2/5, 66, 79	18 / -	63 / 77	19 / 23
15	Cplx8	1/1, 77, 91	2 / -	82 / 84	16 / 16
16	Fir16	1/1, 77, 91	2 / -	81 / 81	17 / 17
17	cosine2	31/8, 81, 91	43 / -	37 / 80	20 / 20
18	FilterRGB	2/1, 84, 97	3 / -	83 / 86	14 / 14
19	collapse_pyr	6/9, 105, 122	22 / -	59 / 79	19 / 21
20	interpolate	48/4, 108, 104	54 / -	46 / 100	0 / 0
21	w_bmp_head	38/25, 110, 92	61 / -	32 / 92	7 / 8
22	matmul	24/4, 116, 124	25 / -	65 / 90	10 / 10
23	invert_matrix	77/16, 357, 378	29 / -	61 / 90	10 / 10
	avg		34 / -	56 / 89	10 / 11

Table 5.3 shows dataflow proprieties, including I/O, nodes, edges, and the three edge categories. For instance, the first benchmark *mac* has 3 inputs, 1 output, 11 nodes, 11 edges, where 36.4% are I/O, 54.5% are $first$' edges, and 9.3% are *reconv*' edges in the YOTT approach. There is no I/O category for YOTO, and 90.9% and 9.1% of the edges are classified in the category $first$, and *reconv*, respectively. *Reconv* = 0% means that there are only trees, a typical design pattern for operations such as reduction tree of a convolution operation. The last row shows the average in each

edge category. Considering YOTO, there are 10% of reconvergent edges and 90% in the first edge category. Inside the first category, 34% are I/O edges. Therefore, the YOTT annotations explore I/O and reconvergent annotations that comes to the average total $34\% + 10\% = 44\%$ of the edges to reach a better solution. In the next sections, we will use the benchmark numbers shown in the first column in Table 5.3.

Note that in the following graphs that the X-axis lists benchmarks 1 through 23 as related to column 1 of Table 5.3. The ordering is based on the number of nodes from smallest (1,$|V| = 11$) to largest (23,$|V| = 357$).

5.4.4 Percentage of Optimal Edges

We evaluate the placement quality by observing the percentage of optimal edge mappings. An edge is optimal when it requires only one wire segment to route, i.e., each *placed_node* and *target_node* are successfully placed as neighbors. A placement is optimal if all edges have an optimal cost. Moreover, an optimal placement requires neither routing nor timing.

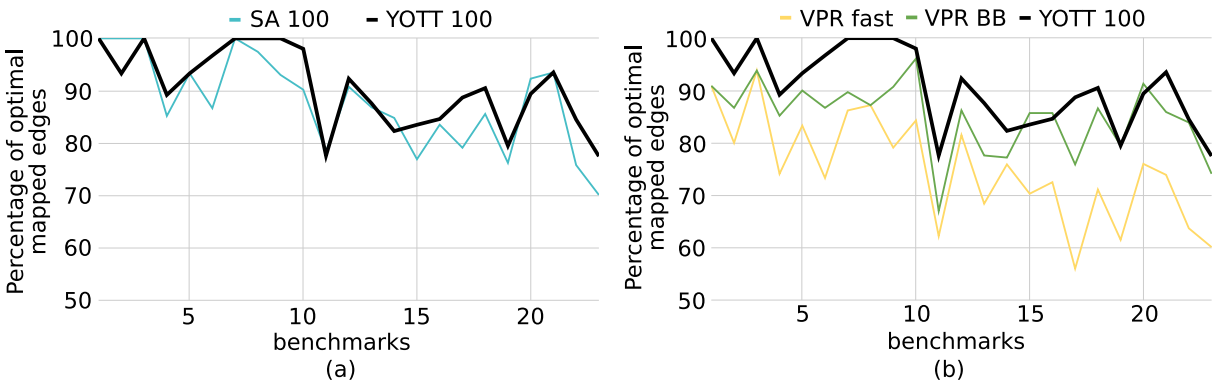

Figura 5.15: Optimal Edges in percentage for (a): SA-100 and YOTT-100; (b) VPR BB and Fast, and and YOTT-100.

Figure 5.15 shows the percentage of optimal edges (y-axis) for the benchmarks (x-axis) after the YOTT-100 and three simulated annealing placements: SA-100, VPR BB, and VPR fast. YOTT-100 outperforms all evaluated SA approaches, and it reaches 22% of optimal placement. SA-100 reaches only 18% of optimal mapping, and it is 60.6% slower than YOTT-100. Moreover, more than 90% of the edges are optimal in 13 out of 23 benchmarks. Finally, it is important to highlight that the routing is already finalized when an edge is optimal. Therefore, YOTT-100 placement solved 90.5% of the routing problems. Moreover, although VPR in fast mode is twice as fast as VPR in bounding box mode (BB), the placement quality degrades as shown in Figure 5.15(b). VPR fast has on average 75.0% of optimal edges, while VPR BB, 84.9%. Furthermore, YOLT-100 is 9.7 and 4.0$\times$ faster than VPR BB and VPR fast, respectively.

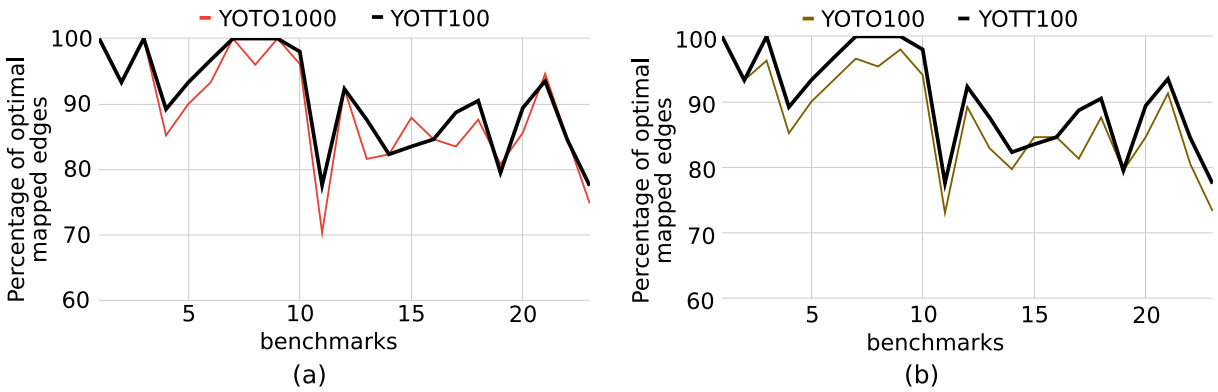

Figura 5.16: Optimal Edges in percentage for YOTT-100 versus (a) YOTO-1000; (b) YOTO-100.

Figure 5.16 shows the percentage of optimal edges for YOTT-100, YOTO-100 and -1000. YOTT-100 outperforms YOTO-100 for all evaluated benchmarks, and it also outperforms YOTO-1000 in 21 of 23 benchmarks, and it is 2.3 $\times$ faster than YOTO-1000. Therefore, by improving the traversal strategy by using the annotation guidelines, even by performing 10$\times$ fewer traversals than YOTO-1000, it improves the placement quality and the execution time performance. Finally, regarding the percentage of optimal edges, YOTT improves both the YOTO performance and SA quality.

5.4.5 Execution Time and Number of Swaps

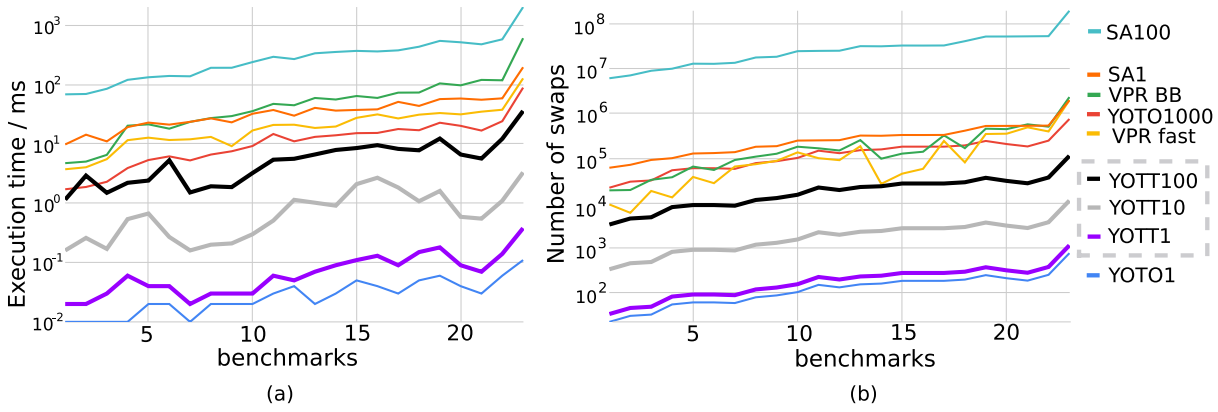

Figura 5.17: Placement Execution Time: SA-based, YOTO, and YOTT. Low is better.

Figure 5.17(a) depicts the execution time in log scale for all evaluated placements. YOTO and YOTT approaches are orders of magnitude faster than SA-based approaches. YOTT fills the gap between a single traversal and SA approaches. YOTT-100 is still faster than YOLO-1000 and provides better results. The average execution time is around 1 millisecond. Therefore it is appropriate for run-time embedded in Just-in-Time compiler or dynamic mapping approaches.

Simulated annealing placement explores the search space by performing graph

node swaps. Figure 5.17(b) shows the total number of SA-based placement in swaps per node. On average, VPR BB performs 2935 swaps per node. Moreover, the number of swaps per node grows linearly with the graph size, and consequently, the execution time is shown in Figure 5.17(a). On the other hand, YOTO and YOTT execution time grows as a function of the graph size and the number of instances, where YOTT-100 is 8.6× slower than YOTT-10. We propose a simple metric to enable comparisons between traversal and simulated annealing heuristics regarding the search space exploration. We observe that YOTO and YOTT, on average, evaluates two and three-node/cell visits during the placement, respectively. Therefore, YOLT-10 and -100 evaluate, respectively, 30 and 300 cells per node. In Figure 5.17 we observe the execution time reduction is a direct consequence of our proposed guided solution search space. YOTT improves the mapping quality and replaces 3000 swaps from VPR BB with 300 and 30 tries to achieve an approximated 10× and 100× speed-up with better quality results. Compared to YOTO-1000, YOTT-100 on average 6.7× less node/cell visits during the placement, and it still improves quality.

5.4.6 Delay Mismatch

In a fully pipelined CGRA (FP-CGRA) (Cong et al., 2014; Nowatzki et al., 2018b), elastic FIFO guarantees the correctness by ensuring data will arrive at the node from all edges at the correct clock cycle. We evaluate YOTO, YOTT, and SA-based approaches concerning this metric, where routing is performed before the FIFO scheduling to reduce the execution time to the order of milliseconds. Figure 5.18 shows the maximum FIFO size for SA-100, VPR BB, VPR fast, and YOTT-100. SA-100 and YOTT-100 reach the best results in FIFO reduction in comparison to all evaluated placements. In 10 and 8 of 23 benchmarks, YOLT-100 and SA-100 reaches the optimal solution and do not require any internal FIFOs, respectively. As already mentioned, YOTT-100 is 60.6× faster than SA-100. In 21 and 22 of 23 benchmarks, the maximal FIFO size is 2 for YOTT-100 and SA-100, respectively, and graph size up to 116 nodes. In comparison to VPR, YOTT-100 is significantly better as shown in Figure 5.18(b). Moreover, YOTT outperforms the hybrid approach proposed in (Nowatzki et al., 2018b), which does not scale for graphs with more than 20 nodes. The execution time is greater than 10 seconds if the maximal FIFO size is bound in 3 and the exact approaches such as CGRA-ME (Walker and Anderson, 2019a) and SAT-solvers (Donovick et al., 2019), which do not scale for graphs with more than 30 nodes. Compared to single traversal approaches, YOTT-100 outperforms YOTO-1000 for all evaluated benchmarks, as shown in Figure 5.19(a). In comparison to YOTO-100, by evaluating the same number of traversals, the YOTT guidelines show better performance, as shown in Figure 5.19(b). Therefore, YOTT better exploits the graph proprieties by using the

annotation guidelines.

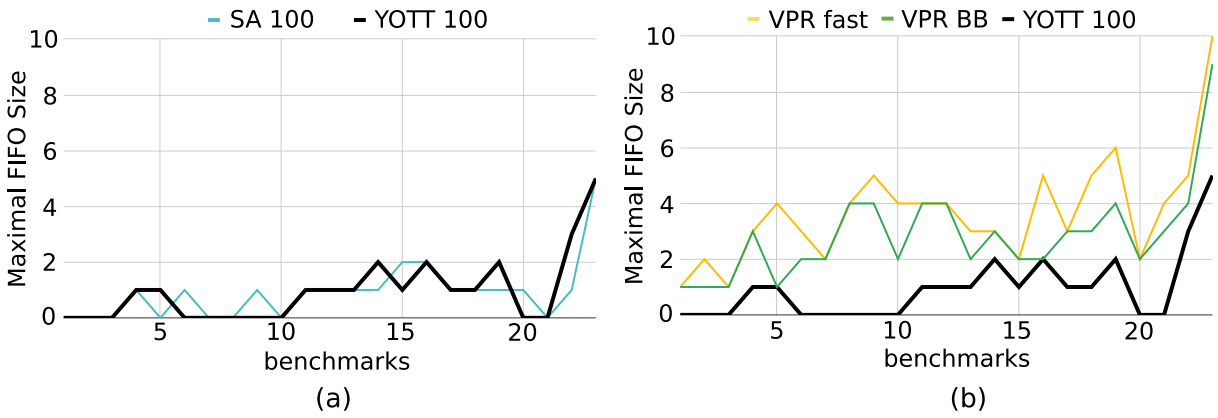

Figura 5.18: Maximal FIFO size for VPR BB and Fast, YOTT-100 and SA-100. Low is better.

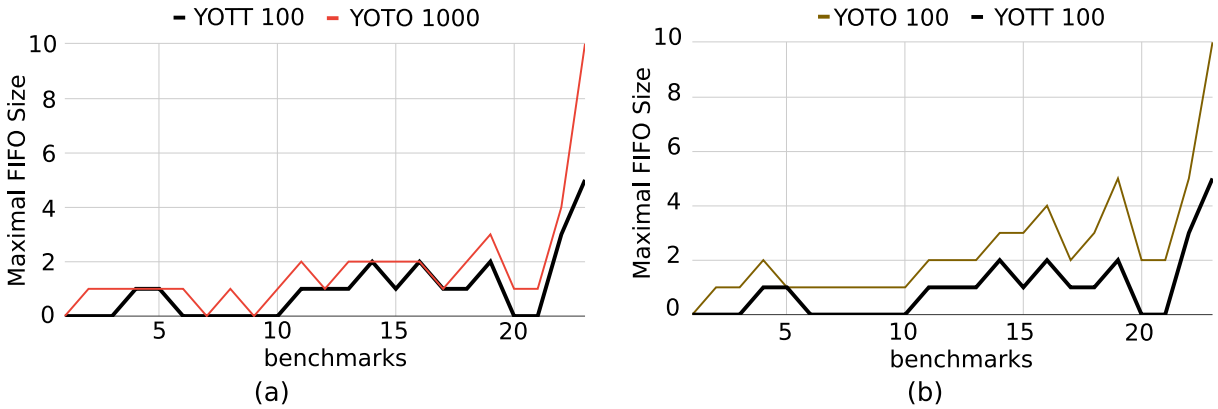

Figura 5.19: Maximum FIFO Size for: YOTO-100 and -1000, YOTT-100. Low is better.

5.4.7 Reconvergent Annotation

We evaluated the impact of reconvergent annotation, considering the maximal FIFO size for the YOTT-10, as shown in Figure 5.20(b). We observe the reduction in the FIFO size due to information with the reconvergent annotation (RA). The execution time increases as shown in Figure 5.20(a).

5.5 Related Work

Parallel designs represented as dataflow graphs can efficiently execute on spatial architectures to process stream data (Weng et al., 2020). Therefore, our approach solves three problems: scheduling, placement, and routing. This space is the focus of our work, and we present related work for CGRA mapping, which is tied to architecture choice.

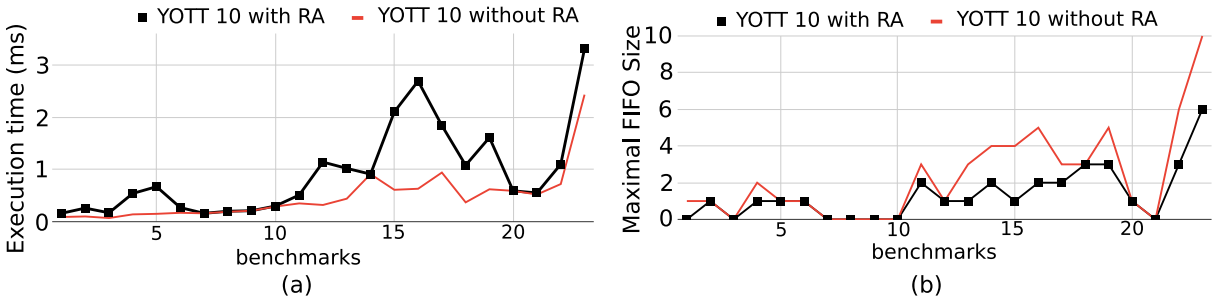

Figura 5.20: Maximal FIFO size with/without Reconvergent Annotation.

CGRAs could be mapped by using time-multiplexed (Hamzeh et al., 2013; Chen and Mitra, 2014b; Karunaratne et al., 2019) or fully pipelined architectures (Cong et al., 2014; Nowatzki et al., 2018b). Most previous work in time-multiplexed architectures has concentrated on 4x4 arrays, limiting the peak performance in 16 computations per cycle. Spatial mapping in large CGRA could significantly improve the peak performance by adding FIFOs to balance all paths (Cong et al., 2014; Nowatzki et al., 2018b).

One way to simplify these algorithmic problems is to reduce the architecture size. Modulo scheduling (Walker and Anderson, 2019a; Karunaratne et al., 2019) adds a temporal component to map large dataflow onto small architectures. However, it does not reduce the problem complexity. Modulo scheduling mapping is equivalent to a spatial graph modeled by a Modulo Routing Resource Graph (MRRG) (Karunaratne et al., 2019). The MRRG represents the temporal component as a spatial one. Furthermore, this increases the architecture cost by adding multiplexer capability, instruction memories, and register files for each architecture cell. Moreover, time-multiplexed (Hamzeh et al., 2013) also requires registers and efficient placement and routing to implement the modulo scheduling, which is still the major challenge (Karunaratne et al., 2019).

Recent work (Nowatzki et al., 2018b; Karunaratne et al., 2019; Walker and Anderson, 2019a; Wijerathne et al., 2021) still shows considerable execution time for small size dataflow graphs, which require seconds to minutes to map an input graph with less than 100 nodes. The additional CGRA challenge compared to VLSI/FPGA mapping is the scheduling issue where all paths should be equalized, which is a necessary condition to guarantee the correct pipeline functioning. The significant advantage of spatial CGRA is that it executes the entire graph in one cycle in a pipeline fashion and could execute 50-150 operations per cycle.

Exact solutions such as integer linear programming (ILP) approach (Walker and Anderson, 2019a) or SAT-solvers (Donovick et al., 2019) do not scale. YOTO traversal (Canesche et al., 2020) is five orders of magnitude faster than CGRA-ME ILP (Walker and Anderson, 2019a). An edge-centric scheduler (Park et al., 2008) prioritizes

simple edges routing over high-fanout edges. Despite that, the execution time consumes a few seconds. Meta-heuristics, such as SA, that include scheduling are also time consuming (Park et al., 2008; Chen and Mitra, 2014b; Walker and Anderson, 2019a). Nevertheless, it is possible to improve these approaches by decoupling placement, routing and scheduling. Also, it should have an efficient placement implementation such as VPR (Luu et al., 2011; Murray et al., 2020). Graph mirror approach (Chen and Mitra, 2014b) simplifies the mapping. However, the average execution time is in the order of 100 milliseconds. Despite polynomial time complexity, graph traversal approaches (Lai et al., 2005; Ferreira et al., 2007; Canesche et al., 2020) degrade the mapping quality, and they employ multiple executions to improve quality while increasing execution time. YOTT enhances placement solution exploration by applying rules based on the dataflow graph proprieties, which leads to better solutions in less than one millisecond, approaching viable for CGRA just-in-time compilation.

5.6 Conclusion

This work introduces a novel *"you only traversal twice"* YOTT approach with placement guidance that pushes the state-of-the-art for CGRA placement, improving quality to that of a Simulated Annealing approach while maintaning the execution time low, close to the simple traversal approaches execution times. YOTT annotates graph properties on the first pass that are later exploited in the following placement step. Our placement guidance adds two new annotation techniques, I/O and reconvergent annotations, and introduces two second step heuristics placement approaches, degree matching, and look-ahead placement direction. We have shown that our approach efficiently maps a logical edge to a physical edge in a more deterministic way for dataflow graphs at the instruction-level. Future work can include considering the problem of mapping dataflows on heterogeneous architectures, aiming an overall cost reduction.

Capítulo 6

Conclusão

Nesta dissertação foram apresentados dois trabalhos em problemas de P&R que abordam travessia em grafos em CGRAs, melhorando a qualidade e o tempo de execução. O algoritmo de travessia (YOTO) baseia-se na abordagem gulosa de posicionamento, enquanto o algoritmo YOTT utiliza as anotações de propriedades do grafo para serem exploradas na etapa posterior.

No Capítulo 4 mostrou-se que o algoritmo traversal usando a técnica modulo *scheduling* com II=2 teve tempo de execução na ordem de 5 vezes menor do que os algoritmos SA e ILP do CGRA-ME usando a mesma arquitetura, além de retornar soluções ótimas para todos os benchmarks. Além disso, foi apresentada uma nova travessia no grafo Zigzag que correlacionam as múltiplas saídas em comparação com outras travessias: profundidade (Ferreira et al., 2007), árvore geradora (Lai et al., 2005) e abordagens híbridas (Nowatzki et al., 2018b). Também foram feitas comparações com o VPR em diferentes cenários. Os resultados mostraram que as a versões ZigZag1 (1 interação), ZigZag1000 (1000 interações) e GPUs foram 502.1x, 4.6x e 10.1x mais rápidas que o VPR, respectivamente.

No Capitulo 5 foi introduzido o algoritmo YOTT em que, diferente do algoritmo Traversal, são feitas anotações sobre as propriedades do grafo e que posteriormente são exploradas sobre a etapa de P&R. A orientação de posicionamento adiciona técnicas de anotações baseadas nas restrições de posicionamento para as entradas e saídas e para as reconvergências do grafo. Além disso, introduz novas abordagens de posicionamento em relação ao grau de liberdade de um nó e da direção do posicionamento *look-ahead*. O algoritmo YOTT, utilizando 100 instâncias de solução, reduziu o comprimento de fio em 7%, economizou o número de filas em 2x e acelerou em até 4x o tempo de execução comparado a abordagem YOTO. Em relação ao algoritmo do VPR, YOTT100 chegou a ser 21x mais rápido, 20% mais econômico em comprimento de fio e reduziu 4x o número de filas.

Em geral, os resultados mostraram que ainda é possível ter ganhos expressivos no problema P&R em comparação com os algoritmos do estado da arte. O YOTT ainda pode melhorar em termos de qualidade de solução, podendo-se utilizar outras abordagens em conjunto. Por exemplo, a Tabela 6.1 apresenta os resultados médios

para o conjunto de *benchmarks* (Benchmarks, 2020) do YOTT100 (100 - instâncias de soluções) comparado com o algoritmo de SA (Carvalho et al., 2020) de 10, 100 e 1000 instâncias de soluções. Os resultados mostram que o YOTT foi melhor do que SA10 e SA100. Contudo, comparando entre os algoritmos YOTT100 e o SA1000, os resultados mostram que o SA consegue ser 7% mais econômico no comprimento de fio e 15% em economia no uso de filas em comparação com YOTT100. Entretanto, o tempo da execução do SA é em média 784x maior que o YOTT. Surge então o desafio de melhorar a qualidade sem aumentar o tempo de execução.

Tabela 6.1: Tabela comparativa YOTT versus SA

	Média		
Algoritmo	1-Hop	Fila	Tempo (ms)
YOTT100	1.16	0.96	**6.80**
SA10	1.18	1.78	91.30
SA100	1.17	1.52	619.13
SA1000	**1.08**	**0.83**	5340.87

Como trabalhos futuros, tecnologias modernas como *Quantum-dot Cellular Automata* (QCA) (Fontes et al., 2018; Fontes Junior et al., 2018) e *Nanomagnetic Logic* (NML) (Formigoni et al., 2020) vêm ganhando espaço para substituir a atual tecnologia baseada em silício para a fabricação de circuitos integrados. Uma abordagem utilizando o algoritmo de YOTT em tecnologias QCA e NML podem trazer benefícios como diminuição da área ocupada e diminuição do cruzamento de fios, pois o algoritmo que o YOTT foi moldado para diminuir recursos e obter posicionamentos estratégicos. Contudo, alguns desafios devem ser enfrentados devido as restrições de direção dos fios (zonas de *clock*) nas etapas de posicionamento e roteamento em projetos de circuitos QCA e NML. Além disso, devido a falta do recurso de filas para balanceamentos nessas tecnologias, será necessário o uso de técnicas de fios mais longos na etapa de escalonamento, podendo acontecer um aumento nos cruzamento de fios.

Outro trabalho interessante seria a aplicação de CGRAs em redes reguladoras de genes (GRN). Uma GRN pode ser modelada como um grafo de equações booleanas (Rosa et al., 2019), onde um vértice representa se o extrator está ativo ou não e as arestas seriam dependências das equações. Contudo a estrutura do grafo é diferente, pois as topologias das redes se diferem do tamanho de conexão dos problemas de grafos em CGRAs, pois muitos vértices possuem grau de conexões pequenos e poucos tem grau de conexões grandes. Além disso, a cada passo de simulação ocorre a redução onde é necessário capturar o estado de todos os extratores. Um modelo de CGRA iria precisar de duas redes: global e local. Uma rede global seria para capturar os estados de todos os extratores e uma rede local seria somente para comunicar com os vizinhos. Como seria necessário ter várias instâncias da rede rodando em paralelo,

a minimização do espaço ocupado em um CGRA seria de extrema importância nesse caso, pois o espaço de busca em uma GRN é grande. Por exemplo, em um grafo de atrator síncrono não há muito o que ser feito, pois os processos acontecem em um tempo determinado. Contudo, em um grafo de atrator assíncrono é possível fazer flexibilização no grafo, como fixar alguns vértices, como se eles não existissem no grafo original ou remover vértices que não alteram a rede de genes. Vale ressaltar que GPUs são ineficientes em GRN devido ao grande número de dependências das equações booleanas, mas em FPGAs são ótimas e nesse caso precisam de um CGRA, uma vez que o grafo é gerado dinamicamente e terá conhecimento de quais pontos serão observados, quais pontos foram fixados e quais pontos serão desprezados.

Enfim, trabalhos utilizando técnicas de predição para acelerar a execução em fluxo de controle em CGRAs são um caminho promissor na área (Han et al., 2013). Além disso, o uso de técnicas de aprendizado de máquina em P&R (Liu et al., 2018) em conjunto com redes neurais profundas (Lin et al., 2020b) podem ajudar o algoritmo YOTT a diminuir os recursos gastos em um CGRA e otimiza-lo para ter melhores posicionamentos.

Referências Bibliográficas

Ababei, C. (2009). Speeding up fpga placement via partitioning and multithreading. *Int Journal of Reconfigurable Computing*.

Al-hyari, A., Shamli, A., Martin, T., Areibi, S., and Grewal, G. (2020). An adaptive analytic fpga placement framework based on deep-learning. In *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*, pages 3–8.

Alhyari, A., Shamli, A., Abuwaimer, Z., Areibi, S., and Grewal, G. (2019). A deep learning framework to predict routability for fpga circuit placement. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 334–341. IEEE.

Bansal, N., Gupta, S., Dutt, N., Nicolau, A., and Gupta, R. (2004a). Network topology exploration of mesh-based coarse-grain reconfigurable architectures. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 1, pages 474–479. IEEE.

Bansal, N., Gupta, S., Dutt, N., Nicolau, A., and Gupta, R. (2004b). Network topology exploration of mesh-based coarse-grain reconfigurable architectures. In *Design, Automation and Test in Europe*. IEEE.

Baumgarte, V., Ehlers, G., May, F., Nückel, A., Vorbach, M., and Weinhardt, M. (2003). Pact xpp—a self-reconfigurable data processing architecture. *the Journal of Supercomputing*, 26(2):167–184.

Becker, T., Mencer, O., and Gaydadjiev, G. (2016). Spatial programming with openspl. In *FPGAs for Software Programmers*, pages 81–95. Springer.

Benchmarks (2020). Dataflow graph benchmarks. `https://github.com/canesche/benchmarks`. Último acesso em: 05/10/2020.

Boutros, A., Yazdanshenas, S., and Betz, V. (2018). You cannot improve what you do not measure: Fpga vs. asic efficiency gaps for convolutional neural network inference. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 11(3):1–23.

Canesche, M., Menezes, M., Carvalho, W., Torres, F., Jamieson, P., Nacif, J. A., and Ferreira, R. (2020). Traversal: A fast and adaptive graph-based placement and routing for cgras. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Carvalho, O., Canesche, M., Reis, L., Torres, F., Jamieson, P., Silva, L., Nacif, J., and Ferreira, R. (2020). A design exploration of scalable mesh-based fullypipelined accelerators. In *2020 International Conference on Field-Programmable Technology (ICFPT)*. IEEE.

Chen, D., Cong, J., and Pan, P. (2006). *FPGA design automation: A survey*. Now Publishers Inc.

Chen, L. and Mitra, T. (2014a). Graph minor approach for application mapping on cgras. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(3):21.

Chen, L. and Mitra, T. (2014b). Graph minor approach for application mapping on cgras. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(3):1–25.

Chen, Y.-H., Emer, J., and Sze, V. (2016). Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*.

Chhugani, J., Satish, N., Kim, C., Sewall, J., and Dubey, P. (2012). Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 378–389. IEEE.

Chin, S. A. and Anderson, J. H. (2018a). An architecture-agnostic integer linear programming approach to cgra mapping. In *Design Automation Conference (DAC)*. IEEE.

Chin, S. A. and Anderson, J. H. (2018b). An architecture-agnostic integer linear programming approach to cgra mapping. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6.

Chin, S. A., Sakamoto, N., Rui, A., Zhao, J., Kim, J. H., Hara-Azumi, Y., and Anderson, J. (2017a). Cgra-me: A unified framework for cgra modelling and exploration. In *Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE.

Chin, S. A., Sakamoto, N., Rui, A., Zhao, J., Kim, J. H., Hara-Azumi, Y., and Anderson, J. (2017b). Cgra-me: A unified framework for cgra modelling and exploration. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 184–189. IEEE.

Cong, J., Huang, H., Ma, C., Xiao, B., and Zhou, P. (2014). A fully pipelined and dynamically composable architecture of cgra. In *International Symposium on Field-Programmable Custom Computing Machines*. IEEE.

Cong, J., Romesis, M., and Xie, M. (2003). Optimality and stability study of timing-driven placement algorithms. In *Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, ICCAD '03, pages 472–.

Coole, J. and Stitt, G. (2010). Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *Int. Conf. on Hardware/software codesign and system synthesis*. ACM.

Coole, J. and Stitt, G. (2015). Adjustable-cost overlays for runtime compilation. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 21–24. IEEE.

da Silva, L. B., Almeida, D., Nacif, J. A. M., Sánchez-Osorio, I., Hernández-Martínez, C. A., and Ferreira, R. (2017). Exploring the dynamics of large-scale gene regulatory networks using hardware acceleration on a heterogeneous cpu-fpga platform. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–7. IEEE.

Da Silva, M. V., Ferreira, R., Garcia, A., and Cardoso, J. M. (2006). Mesh mapping exploration for coarse-grained reconfigurable array architectures. In *2006 IEEE International Conference on Reconfigurable Computing and FPGA's (ReConFig 2006)*, pages 1–10. IEEE.

Dave, S., Balasubramanian, M., and Shrivastava, A. (2018). Ramp: Resource-aware mapping for cgras. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE.

Donovick, C., Mann, M., Barrett, C., and Hanrahan, P. (2019). Agile smt-based mapping for cgras with restricted routing networks. In *Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE.

Eguro, K. (2002). *RaPiD-AES: Developing an encryption-specific FPGA architecture*. PhD thesis, University of Washington.

Ferreira, R., Canesche, M., Coelho, K., and Nacif, J. (2018). Minimum switching networks. In *2018 VIII Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 225–230. IEEE.

Ferreira, R., Cardoso, J. M., Toledo, A., and Neto, H. C. (2005). Data-driven regular reconfigurable arrays: design space exploration and mapping. In *International Workshop on Embedded Computer Systems*, pages 41–50. Springer.

Ferreira, R., Damiany, A., Vendramini, J., Teixeira, T., and Cardoso, J. M. (2009). On simplifying placement and routing by extending coarse-grained reconfigurable arrays with omega networks. In *International Workshop on Applied Reconfigurable Computing*, pages 145–156. Springer.

Ferreira, R., Duarte, V., Meireles, W., Pereira, M., Carro, L., and Wong, S. (2013a). A just-in-time modulo scheduling for virtual coarse-grained reconfigurable architectures. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 188–195. IEEE.

Ferreira, R., Garcia, A., Teixeira, T., and Cardoso, J. M. (2007). A polynomial placement algorithm for data driven coarse-grained reconfigurable architectures. In *IEEE Comp Soc Annual Symposium on VLSI*.

Ferreira, R., Rocha, L., Santos, A., Nacif, J., Wong, S., and Carro, L. (2013b). 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.

Ferreira, R., Rocha, L., Santos, A., Nacif, J., Wong, S., and Carro, L. (2013c). A run-time graph-based polynomial placement and routing algorithm for virtual fpgas. In *Int Conf on Field programmable Logic and Applications (FPL)*.

Ferreira, R., Rocha, L., Santos, A. G., Nacif, J. A., Wong, S., and Carro, L. (2015). A runtime fpga placement and routing using low-complexity graph traversal. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 8(2):1–16.

Ferreira, R., Vendramini, J. G., Mucida, L., Pereira, M. M., and Carro, L. (2011a). An fpga-based heterogeneous coarse-grained dynamically reconfigurable architecture. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, pages 195–204.

Ferreira, R. S., Cardoso, J. M., Damiany, A., Vendramini, J., and Teixeira, T. (2011b). Fast placement and routing by extending coarse-grained reconfigurable arrays with omega networks. *Journal of Systems Architecture*, 57(8):761–777.

Fontes, G., Silva, P. A. R., Nacif, J. A. M., Neto, O. P. V., and Ferreira, R. (2018). Placement and routing by overlapping and merging qca gates. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE.

Fontes Junior, G. S. et al. (2018). Explorando o espaço de soluções no posicionamento e roteamento de células qca no esquema de clock use.

Formigoni, R. E., Vieira, L. L. A., Neto, O. P. V., Ferreira, R., and Nacif, J. A. M. (2020). Evaluating nanomagnetic logic circuit layouts using different clock schemes. *Analog Integrated Circuits and Signal Processing*, pages 1–14.

Friedman, S., Carroll, A., Van Essen, B., Ylvisaker, B., Ebeling, C., and Hauck, S. (2009). Spr: an architecture-adaptive cgra mapping tool. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 191–200. ACM.

Gao, X. (2020). *Next Generation Datacenter Architecture*. PhD thesis, Ph. D. Dissertation. EECS Department, University of California, Berkeley ....

Georgis, G., Lentaris, G., and Reisis, D. (2019). Acceleration techniques and evaluation on multi-core cpu, gpu and fpga for image processing and super-resolution. *Journal of Real-Time Image Processing*, 16(4):1207–1234.

Goeders, J. B., Lemieux, G. G., and Wilton, S. J. (2011). Deterministic timing-driven parallel placement by simulated annealing using half-box window decomposition. In *Int Conf on Reconfigurable Computing and FPGAs*. IEEE.

Goldie, A. and Mirhoseini, A. (2020). Placement optimization with deep reinforcement learning. In *Proceedings of the 2020 International Symposium on Physical Design*, pages 3–7.

Gurobi (2019). Integer program solver. `http://www.gurobi.com/`. Último acesso em: 08/10/2019.

HajiRassouliha, A., Taberner, A. J., Nash, M. P., and Nielsen, P. M. (2018). Suitability of recent hardware accelerators (dsps, fpgas, and gpus) for computer vision and image processing algorithms. *Signal Processing: Image Communication*, 68:101–119.

Hamzeh, M., Shrivastava, A., and Vrudhula, S. (2013). Regimap: register-aware application mapping on coarse-grained reconfigurable architectures (cgras). In *Design Automation Conference DAC*. ACM.

Han, K., Ahn, J., and Choi, K. (2013). Power-efficient predication techniques for acceleration of control flow execution on cgra. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(2):1–25.

Han, T. D. and Abdelrahman, T. S. (2011). Reducing branch divergence in gpu programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, page 3. ACM.

Hartenstein, R. (2001). A decade of reconfigurable computing: a visionary retrospective. In *Proceedings design, automation and test in Europe. Conference and exhibition 2001*, pages 642–649. IEEE.

Huang, Y., Ienne, P., Temam, O., Chen, Y., and Wu, C. (2013). Elastic cgras. In *Int Symposium on Field programmable gate arrays*. ACM.

Ijaz, Q., Bourennane, E.-B., Bashir, A. K., and Asghar, H. (2020). Revisiting the high-performance reconfigurable computing for future datacenters. *Future Internet*, 12(4):64.

Karunaratne, M., Mohite, A. K., Mitra, T., and Peh, L.-S. (2017). Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6.

Karunaratne, M., Wijerathne, D., Mitra, T., and Peh, L.-S. (2019). 4d-cgra: Introducing branch dimension to spatio-temporal application mapping on cgras. In *ICCAD*, pages 1–8.

Kim, J., Hwang, I., Kim, Y.-H., and Moon, B.-R. (2011a). Genetic approaches for graph partitioning: a survey. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 473–480.

Kim, S., Park, Y.-H., Kim, J., Kim, M., Lee, W., and Lee, S. (2015). Flexible video processing platform for 8k uhd tv. In *Hot Chips Symposium*, page 1.

Kim, Y., Lee, J., Shrivastava, A., and Paek, Y. (2011b). Memory access optimization in compilation for coarse-grained reconfigurable architectures. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 16(4):1–27.

Kojima, T., Doan, N. A. V., and Amano, H. (2020). Genmap: A genetic algorithmic approach for optimizing spatial mapping of coarse-grained reconfigurable architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(11):2383–2396.

Koren, I., Mendelson, B., Peled, I., and Silberman, G. M. (1988). A data-driven vlsi array for arbitrary algorithms. *Computer*, 21(10):30–43.

Lai, Y.-T., Lai, H.-Y., and Yeh, C.-N. (2005). Placement for the reconfigurable datapath architecture. In *IEEE Symp on Circuits and Systems*.

Lattner, C. and Adve, V. (2004). Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE.

Lee, J. E., Choi, K., and Dutt, N. (2008). Evaluating memory architectures for media applications on coarse-grained reconfigurable architectures. *International Journal of Embedded Systems*, 3(3):119–127.

Lin, Y., Jiang, Z., Gu, J., Li, W., Dhar, S., Ren, H., Khailany, B., and Pan, D. Z. (2020a). Dreamplace: Deep learning toolkit-enabled gpu acceleration for modern vlsi placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Lin, Y., Jiang, Z., Gu, J., Li, W., Dhar, S., Ren, H., Khailany, B., and Pan, D. Z. (2020b). Dreamplace: Deep learning toolkit-enabled gpu acceleration for modern vlsi placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Liu, D., Yin, S., Luo, G., Shang, J., Liu, L., Wei, S., Feng, Y., and Zhou, S. (2018). Data-flow graph mapping optimization for cgra with deep reinforcement learning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(12):2271–2283.

Liu, D., Yin, S., Luo, G., Shang, J., Liu, L., Wei, S., Feng, Y., and Zhou, S. (2019a). Data-flow graph mapping optimization for cgra with deep reinforcement learning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Liu, L., Zhu, J., Li, Z., Lu, Y., Deng, Y., Han, J., Yin, S., and Wei, S. (2019b). A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications. *ACM Computing Surveys (CSUR)*, 52(6):1–39.

Luu, J., Kuon, I., Jamieson, P., Campbell, T., Ye, A., Fang, W. M., Kent, K., and Rose, J. (2011). Vpr 5.0: Fpga cad and architecture exploration tools with single-driver routing, heterogeneity and process scaling. *Trans. on Reconfigurable Technology and Systems (TRETS)*, 4.

Maarouff, D., Shamli, A., Martin, T., Grewal, G., and Areibi, S. (2020). A deep-learning framework for predicting congestion during fpga placement. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 138–144. IEEE.

Mehta, G., Patel, K., and Pollard, N. S. (2015). On fast iterative mapping algorithms for stripe based coarse-grained reconfigurable architectures. *International Journal of Electronics*, 102(1):3–17.

Mehta, G., Patel, K. K., Parde, N., and Pollard, N. S. (2013). Data-driven mapping using local patterns. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(11):1668–1681.

Mei, B., Berekovic, M., and Mignolet, J. (2007). Adres & dresc: Architecture and compiler for coarse-grain reconfigurable processors. In *Fine-and coarse-grain reconfigurable computing*, pages 255–297. Springer.

Mei, B., Vernalde, S., Verkest, D., De Man, H., and Lauwereins, R. (2003). Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In *International Conference on Field Programmable Logic and Applications*, pages 61–70. Springer.

Mirhoseini, A., Goldie, A., Yazgan, M., Jiang, J., Songhori, E., Wang, S., Lee, Y.-J., Johnson, E., Pathak, O., Bae, S., et al. (2020). Chip placement with deep reinforcement learning. *arXiv preprint arXiv:2004.10746*.

Murray, K. E., Petelin, O., Zhong, S., Wang, J. M., Eldafrawy, M., Legault, J.-P., Sha, E., Graham, A. G., Wu, J., Walker, M. J., et al. (2020). Vtr 8: High-performance cad and customizable fpga architecture modelling. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 13(2):1–55.

Nadel, A. (2017). A correct-by-decision solution for simultaneous place and route. In *International Conference on Computer Aided Verification*, pages 436–452. Springer.

Nicol, C. (2017). A coarse grain reconfigurable array (cgra) for statically scheduled data flow computing. *Wave Computing White Paper*.

Nowatzki, T., Ardalani, N., Sankaralingam, K., and Weng, J. (2018a). 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.

Nowatzki, T., Ardalani, N., Sankaralingam, K., and Weng, J. (2018b). Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign. In *Int. Conf. on Parallel Architectures and Compilation Techniques PACT*.

Oliveira, W., Canesche, M., Reis, L., Nacif, J., and Ferreira, R. (2020). Design exploration of machine learning data-flows onto heterogeneous reconfigurable hardware. In *Anais do XXI Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 119–130. SBC.

Park, H., Fan, K., Mahlke, S. A., Oh, T., Kim, H., and Kim, H.-s. (2008). Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 166–176.

Park, H., Park, Y., and Mahlke, S. (2009a). Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *Int. Symposium on Microarchitecture*. ACM.

Park, Y., Park, H., and Mahlke, S. (2009b). Cgra express: accelerating execution using dynamic operation fusion. In *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 271–280.

Pasandi, G. and Pedram, M. (2019). A dynamic programming-based, path balancing technology mapping algorithm targeting area minimization. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE.

Pattison, R., Fobel, C., Grewal, G., and Areibi, S. (2015). Scalable analytic placement for fpga on gpgpu. In *Int Conf on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE.

Podobas, A., Sano, K., and Matsuoka, S. (2020). A survey on coarse-grained reconfigurable architectures from a performance perspective. *IEEE Access*, 8:146719–146743.

Prabhakar, R., Zhang, Y., Koeplinger, D., Feldman, M., Zhao, T., Hadjis, S., Pedram, A., Kozyrakis, C., and Olukotun, K. (2017). Plasticine: A reconfigurable architecture for parallel patterns. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 389–402. IEEE.

Rau, B. R. (1996). Iterative modulo scheduling. *International Journal of Parallel Programming*, 24(1):3–64.

Robič, B. and Šilc, J. (1994). Using parallel simulated annealing in the mapping problem. In *International Conference on Parallel Architectures and Languages Europe*, pages 797–800. Springer.

Rosa, W., Baranda, H., Canesche, M., Menezes, M., Bragança, L., Magalhaes, S., Nacif, J. A., and Ferreira, R. (2019). Simulação de redes reguladoras de genes com lógica booleana e limiar em plataformas alto desempenho. In *Anais do XX Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 334–345. SBC.

Schafer, B. C. and Wang, Z. (2019). High-level synthesis design space exploration: Past, present and future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Silva, L. B., Canesche, M., Ferreira, R., and Nacif, J. A. M. (2020). Hpcgra-an orthogonal designed cgra generator for high performance spatial accelerators. *XXI Simpósio em Sistemas Computacionais de Alto Desempenho*.

Silva, L. B. D., Ferreira, R., Canesche, M., Menezes, M. M., Vieira, M. D., Penha, J., Jamieson, P., and Nacif, J. A. M. (2019). Ready: A fine-grained multithreading overlay framework for modern cpu-fpga dataflow applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–20.

Silva, M. V. d. et al. (2006). Exploração do espaço de projeto de arquiteturas reconfiguráveis em arranjos.

Tan, C., Xie, C., Li, A., Barker, K. J., and Tumeo, A. (2020). Opencgra: An open-source unified framework for modeling, testing, and evaluating cgras. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 381–388. IEEE.

Taras, I. and Anderson, J. H. (2019). Impact of fpga architecture on area and performance of cgra overlays. In *Int Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE.

Taylor, M. B., Kim, J., Miller, J., Wentzlaff, D., Ghodrat, F., Greenwald, B., Hoffman, H., Johnson, P., Lee, J.-W., Lee, W., et al. (2002). The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE micro*, 22(2):25–35.

Tehre, V. and Kshirsagar, R. (2012). Survey on coarse grained reconfigurable architectures. *International Journal of Computer Applications*, 48(16):1–7.

Theis, T. N. and Wong, H.-S. P. (2017). The end of moore's law: A new beginning for information technology. *Computing in Science & Engineering*, 19(2):41.

Thies, W., Karczmarek, M., and Amarasinghe, S. (2002). Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, pages 179–196. Springer.

Trindade, A., Ferreira, R., Nacif, J. A. M., Sales, D., and Neto, O. P. V. (2016). A placement and routing algorithm for quantum-dot cellular automata. In *2016 29th symposium on integrated circuits and systems design (SBCCI)*, pages 1–6. IEEE.

Tukur, A. D., Nzerem, P., Nsan, N., Okafor, I. S., Gimba, A., Ogolo, O., Oluwaseun, A., Andrew, O., et al. (2019). Well placement optimization using simulated annealing and genetic algorithm. In *SPE Nigeria Annual International Conference and Exhibition*. Society of Petroleum Engineers.

University of California, S. B. (2020). Express benchmarks. `https://web.ece.ucsb.edu/EXPRESS/benchmark/`. Último acesso em: 10/12/2020.

Vieira, M., Canesche, M., Bragança, L., Campos, J., Silva, M., Ferreira, R., and Nacif, J. A. (2021). 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.

Walker, M. J. and Anderson, J. H. (2019a). Generic connectivity-based cgra mapping via integer linear programming. In *Int. Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE.

Walker, M. J. and Anderson, J. H. (2019b). 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.

Wang, C. C. and Lemieux, G. G. (2011). Scalable and deterministic timing-driven parallel placement for fpgas. In *Int Symposium on Field programmable gate arrays*. ACM.

Wang, W., Meng, Q., and Zhang, Z. (2017). A survey of fpga placement algorithm research. In *2017 7th IEEE International Conference on Electronics Information and Emergency Communication (ICEIEC)*, pages 498–502. IEEE.

Weng, J., Liu, S., Dadu, V., and Nowatzki, T. (2019). Daegen: A modular compiler for exploring decoupled spatial accelerators. *IEEE Computer Architecture Letters*.

Weng, J., Liu, S., Wang, Z., Dadu, V., and Nowatzki, T. (2020). A hybrid systolic-dataflow architecture for inductive matrix algorithms. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 703–716. IEEE.

Wijerathne, D., Li, Z., Pathania, A., Mitra, T., and Thiele, L. (2021). Himap: Fast and scalable high-quality mapping on cgra via hierarchical abstraction. In *DATE*.

Yin, S., Liu, D., Sun, L., Liu, L., and Wei, S. (2017). Dfgnet: Mapping dataflow graph onto cgra by a deep learning approach. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4. IEEE.

Yoon, J. W., Shrivastava, A., Park, S., Ahn, M., Jeyapaul, R., and Paek, Y. (2008). Spkm: A novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures. In *2008 Asia and south pacific design automation conference*, pages 776–782. IEEE.

Yoon, J. W., Shrivastava, A., Park, S., Ahn, M., and Paek, Y. (2009a). 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.

Yoon, J. W., Shrivastava, A., Park, S., Ahn, M., and Paek, Y. (2009b). 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.

Yu, L., Jinjiang, Y., and Leibo, L. (2020). Efficient fft implementation on a cgra. In *Proceedings of the 2020 5th International Conference on Mathematics and Artificial Intelligence*, pages 1–4.

Zhang, J. and Li, J. (2017). Improving the performance of opencl-based fpga accelerator for convolutional neural network. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 25–34.

Zhang, X. (2019). Software system research in post-moore's law era: a historical perspective for the future. *Science China Information Sciences*, 62(9):196101.

Zhao, Z., Sheng, W., Wang, Q., Yin, W., Ye, P., Li, J., and Mao, Z. (2020). Towards higher performance and robust compilation for cgra modulo scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 31(9):2201–2219.

Zhou, L., Liu, H., and Liu, D. (2013a). A novel cgra architecture and mapping algorithm for application acceleration. In *CCF National Conference on Compujter Engineering and Technology*, pages 218–227. Springer.

Zhou, L., Liu, H., and Zhang, J. (2013b). Loop acceleration by cluster-based cgra. *IEICE Electronics Express*, pages 10–20130506.