Algoritmos de Posicionamento e Roteamento baseados em Travessia de Grafo para Arquiteturas Reconfiguráveis de Grão Grosso (CGRA)

Michael Canesche[1], **Ricardo Ferreira**[1]

[1]Departamento de Informática – Universidade Federal de Viçosa (UFV)
Avenida Peter Henry Rolfs – 36.570-900 – Viçosa – MG – Brazil

{michael.canesche,ricardo}@ufv.br

Resumo. *O hardware reconfigurável é flexível, eficiente energeticamente e oferece alto desempenho para vários domínios de aplicação. Os FPGAs são a tecnologia reconfigurável mais utilizada, entretanto o mapeamento da aplicação é NP-Completo. Os CGRAs simplificam o desenvolvimento de aplicações para hardware reconfigurável e alteram a granularidade das operações do nível de bit para o nível de palavras (8-32 bits). Este trabalho apresenta avanços no estado da arte para o problema de mapeamento em CGRA propondo novos algoritmos de travessia de grafos. As soluções alcançadas são ordens de grandeza mais rápidas sem comprometer a qualidade. Os algoritmos propostos foram* $91\times$ *mais rápidos com uma redução de* $1.7\times$ *nos recursos de balanceamento em comparação com a abordagem com Simulated Annealing.*

1. Introdução

Os FPGAs (*Field-Programmable Gate Array*) são dispositivos semicondutores com uma matriz de blocos lógicos configuráveis interligados por conexões programáveis. Os FPGAs podem ser personalizados para uma implementação de um algoritmo diretamente em hardware, reduzindo o consumo de energia com alto desempenho. Entretanto, o tempo de mapeamento de uma aplicação nos FPGAs demanda de minutos a horas, ou até mesmo dias. Durante o mapeamento, um dos principais desafios é a etapa de posicionamento e roteamento. As arquiteturas reconfiguráveis de grão grosso (CGRAs) reduzem a complexidade ao trabalhar com palavras maiores (8-32 bits) em comparação com os FPGAs que trabalham no nível de bit. Entretanto, os CGRAs também enfrentam desafios [Silva and *et al* 2019], onde a falta de ferramentas dificulta sua popularização. Este trabalho propõe algoritmos para o problema de posicionamento e roteamento (P&R), que é NP-Completo. Vários trabalhos apresentaram soluções exatas e heurísticas como, por exemplo, *Simulated Annealing* [Murray and *et al* 2020], heurísticas de travessia [Ferreira and et al. 2007], particionamento e técnicas de programação linear [Walker and et al. 2019]. Contudo, o tempo de execução, a qualidade da solução e a limitação para o tamanho máximo do grafo de entrada ainda são barreiras para a solução do problema. O P&R para CGRA inclui uma dificuldade adicional em comparação com os FPGAs que é o balanceamento dos caminhos. Portanto, além de posicionar os elementos e interconectá-los (roteamento), deve-se ainda equilibrar o comprimento dos caminhos para que a implementação em pipeline tenha os fluxos de dados sincronizados corretamente. Caso não seja possível encontrar uma solução exata, a inserção de filas de tamanho ajustável nos caminhos desbalanceados resolve o problema. Entretanto, as

filas podem aumentar o custo da solução em 50% ou mais [Nowatzki and *et al* 2018]. Portanto, mesmo para soluções com filas é importante minimizar o uso deste recurso.

2. Caracterização do problema de pesquisa

O mapeamento de um grafo de fluxo de dados em uma arquitetura CGRA envolve três etapas: posicionamento, roteamento e escalonamento. A capacidade de posicionar, rotear e escalonar depende tanto da estrutura física da arquitetura do CGRA quando do algoritmo de mapeamento e da integração entre as suas três etapas [Nowatzki and *et al* 2018]. Considerando o mapeamento estático, pode-se descrever cada etapa como:

- **Posicionamento**: Atribuir uma unidade de processamento para cada nodo do grafo de fluxo de dados;
- **Roteamento**: Atribuir os 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 da aplicação.

O problema pode ser modelado com programação linear inteira (ILP) [Walker and et al. 2019] considerando as restrições das três etapas conjuntamente. Entretanto, a complexidade do problema inviabiliza o uso das técnicas de ILP para grafos com mais de 25 vértices. Para reduzir a complexidade, cada etapa pode ser solucionada separadamente. Primeiro é realizado o posicionamento com alguma função de custo. Em seguida, o algoritmo de roteamento é aplicado, porém pode ser que não seja possível realizar o roteamento caso o posicionamento seja ruim. Sendo possível encontrar uma solução para o roteamento, pode ser que não seja possível ter recursos suficientes para balancear os caminhos e finalizar o escalonamento.

O estado da arte dos algoritmos de posicionamento é baseado na técnica de *Simulated Annealing* (SA) [Murray and *et al* 2020]. A abordagem com SA gera um excelente posicionamento, simplificando as etapas de roteamento e escalonamento, além de garantir a viabilidade de uma solução. Ademais, reduz significativamente o tempo de execução do posicionamento. Entretanto, ainda é possível reduzir e simplificar ainda mais a solução com algoritmos gulosos baseados em travessia de grafos [Ferreira and *et al* 2013]. Para o P&R de FPGAs, o algoritmo guloso de travessia é três ordens de magnitude mais rápido, pois tem complexidade polinomial. Entretanto, a qualidade da solução do SA é melhor que o de travessia.

3. Motivação

Trabalhos recentes mostram que os CGRAs podem ter um consumo de menos de 1mW [Gobieski and et al. 2021]. A arquitetura SNAFU com 36 elementos de processamento consome 81% menos energia que um processador RISC-V de um núcleo e é 9.9$\times$ mais rápido [Gobieski and et al. 2021]. Entretanto, o ambiente SNAFU utiliza o mapeamento com programação inteira que não escala para grafos maiores que 30 operações.

O desafio dessa dissertação foi a investigação de algoritmos de P&R e escalonamento com escalabilidade, qualidade e baixo tempo de execução. As abordagens com SA recorrem à heurística com uma busca probabilística. Entretanto, o SA acaba realizando muitas tentativas na busca da solução. Por outro lado, as abordagens de travessia gulosa podem não alcançar resultados com a qualidade desejada, apesar da redução significativa no tempo execução.

4. Objetivos

O objetivo dessa dissertação é o desenvolvimento de novos algoritmos de mapeamento de grafo de fluxo de dados em CGRAs baseados na abordagem de travessia. Os algoritmos detectam a localidade no grafo de fluxo de dados e transferem esta localidade para gerar um posicionamento de qualidade no CGRA. A qualidade da solução é alcançada com uma exploração inteligente no espaço de busca.

4.1. Objetivos específicos

- Desenvolver novos percursos para os algoritmos de travessia.
- Explorar as propriedades de localidade nos grafos durante travessia para reduzir o espaço de busca, gerando soluções com qualidade e baixo tempo de execução.
- Avaliar se os algoritmos de travessia podem ser mais eficientes em qualidade e em tempo de execução em comparação com as soluções com *Simulated Annealing* e o estado da arte de algoritmos para CGRAs.

5. Contribuições

Esta dissertação tem 8 contribuições relevantes. Primeiro, os trabalhos anteriores com travessia [Ferreira and *et al* 2013] usaram apenas a busca em profundidade e largura. Estes percursos não consideram a correlação entre as múltiplas saídas do grafo de fluxo. Nesta dissertação apresentamos uma nova travessia denominada travessia em ZigZag. Esta travessia é polinomial como as anteriores e considera as correlações, melhorando a qualidade dos resultados. Segundo, durante a travessia do grafo de fluxo de dados e do grafo do CGRA, existem muitas decisões a serem consideradas. Esta dissertação sistematizou a exploração em três eixos: posição inicial no CGRA, percurso no grafo de fluxo de dados e percurso no grafo do CGRA. A terceira contribuição foi a implementação em GPU da exploração em três eixos. Estas três primeiras contribuições foram publicadas no artigo ”Traversal: A fast and adaptive graph-based placement and routing for cgras.” [Canesche et al. 2020]. A quarta contribuição foi o desenvolvimento de um novo algoritmo que realiza duas travessias de complexidade polinomial nos grafos, realizando uma busca guiada e inteligente no espaço de soluções. A dupla travessia detecta recovergência interna dos caminhos e localização dos vértices de entrada e saída. A quinta contribuição foi a poda no espaço de busca da das escolhas no CGRA com a verificação do grau dos vértices e uma pré-validação do posicionamento na proximidade de uma reconvergência, chamada de *lookahead*. O algoritmo de dupla travessia foi publicado no período ACM Transactions on Embedded Computing Systems [Canesche et al. 2021]. A sexta contribuição foram a implementação do roteamento e de escalonamento. Como o posicionamento conecta diretamente 80-90% dos vértices, o roteamento precisa lidar apenas com 10-20% das arestas, que em geral, estão próximas. Estes algoritmos foram utilizados para validar às duas abordagens de posicionamento propostas [Canesche et al. 2020, Canesche et al. 2021]. Além disso, foram utilizados para validar uma nova versão de posicionamento com SA do nosso grupo de pesquisa [Carvalho et al. 2020, Oliveira et al. 2020]. A sétima contribuição foi a validação dos algoritmos de travessia com uma implementação em hardware [Vieira et al. 2021]. Finalmente, como maior contribuição podemos afirmar que os algoritmos de travessia podem gerar soluções melhores que a abordagem baseada em SA para CGRAs e, ao mesmo tempo, reduzirem o tempo do mapeamento.

6. Trabalhos relacionados

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

Um algoritmo baseado em travessia em tempo polinomial $O(|V|^2)$ foi 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 and et al. 2009]. 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 et al. 2019], tem problemas de escalabilidade e tempo de execução da ordem de segundos a minutos para grafos com 20 à 30 nodos. Um algoritmo polinomial baseado em travessia em profundidade foi proposto em [Ferreira and et al. 2007]. Entretanto, o CGRA da arquitetura alvo considera um CGRA assíncrono. Caso algum caminho fique desbalanceado, a vazão será reduzida, pois alguns elementos de processamento irão aguardar mais tempo aguardando os dados. Apesar do mapeamento ser rápido a qualidade é bem inferior às implementações com *Simulated Annealing* (SA) [Carvalho et al. 2020] pois a implementação proposta em [Ferreira and et al. 2007] é gulosa e explora pouco o espaço de solução.

Outros trabalhos procuram utilizar inteligência artificial para resolver problemas de P&R. Em [Liu and *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. Seus resultados mostraram um desempenho comparável na qualidade com as heurísticas DFG-Net [Yin and et al. 2017], Pattern e SPKM [Yoon and et al. 2009]. Além disso, se adapta a diferentes arquiteturas com uma conversão rápida. Contudo, o tempo de execução é elevado. Por exemplo, para um grafo com apenas 21 nodos foram necessários 16 minutos para encontrar a solução. 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 and et al. 2009].

O posicionamento baseado em SA ainda é considerado o estado da arte [Murray and *et al* 2020], porém para CGRA mostramos que a abordagem com travessia é competitiva, melhorando o tempo de execução e a qualidade da execução.

7. Metodologia

Nesta dissertação primeiro avaliamos os algoritmos baseados em travessia [Ferreira and et al. 2007]. Existem muitas possibilidades para a travessia do grafo de fluxo de dados e do grafo da arquitetura. Para sistematizar esta pesquisa no espaço de busca propusemos o algoritmo *Traversal*. 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. A Figura 1(a) apresenta um exemplo de travessia onde começamos pelo nodo A, poderia ser qualquer outro.

Depois, tomamos a decisão de seguir por B. Depois, temos novamente duas opções, onde seguimos por D.

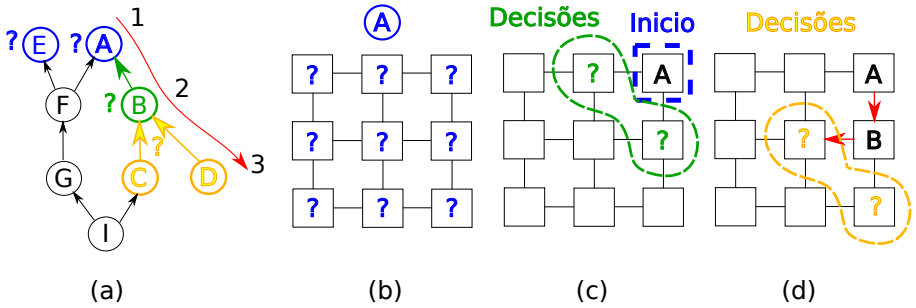

Figura 1. (a) Grafo de fluxo de dados; tomadas de decisão: (b) Nodo inicial; (c) Posicionamento de um novo nodo; (d) Escolha de um novo nodo.

Existem também decisões que devem ser tomadas ao percorrer o grafo da arquitetura. A Figura 1(b) ilustra 9 possibilidades para posicionar o nodo A. Uma vez posicionado A, a Figura 1(c) mostra a próxima decisão. Como B é vizinho de A no grafo de entrada, o algoritmo irá posicionar B em um nodo vizinho de A. O nodo 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, escolhemos a opção ilustrada na Figura 1(d). O próximo passo busca uma posição para o nodo D como vizinho de B. A qualidade da solução dependente diretamente das decisões tomadas durante a travessia do grafo de entrada e do grafo da arquitetura.

7.1. Algoritmo Traversal ou YOTO

Para melhorar a travessia, primeiro investigamos novos percursos. Diferente dos algoritmos tradicionais, esta dissertação propôs a travessia Zigzag percorre tanto na direção saída para entrada ($S \rightarrow E$) tanto no sentido entrada para saída ($E \rightarrow S$) buscando capturar as correlações em grafos com múltiplas saídas. A figura 2 compara uma travessia em profundidade com uma travessia com Zigzag. A Figura 2(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 2(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 2(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.

O percurso Zigzag é apresentado na Figura 2(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

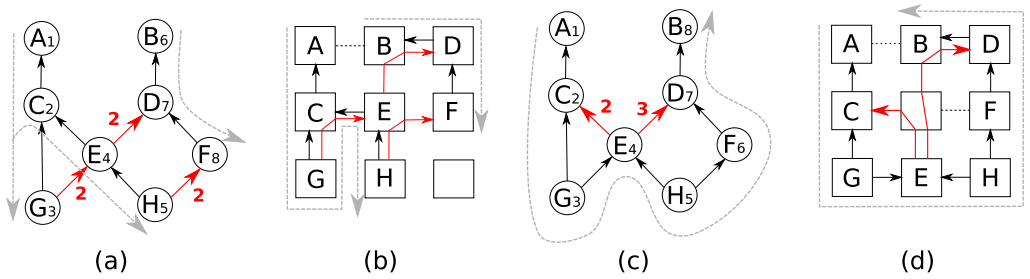

Figura 2. Comparação do Percurso em Profundidade com o Percurso ZigZag: (a) Grafo de Entrada com Travessia em Profundidade; (b) Grafo da Arquitetura em Profundidade; (c) Travessia em ZigZag do Grafo de Entrada; (d) Grafo da Arquitetura percorrido em ZigZag.

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 2(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. Para resolver este problema, implementamos uma exploração nos três eixos: escolha do vértice inicial no grafo de fluxo de dados, escolha da posição inicial no CGRA e escolha do vizinho no CGRA. 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, mesmo com a exploração sistematizada nos três eixos. Na próxima seção apresentamos o algoritmo de dupla travessia ou YOTT que realiza duas travessias no grafo de entrada para capturar e transferir esta localidade.

7.2. Algoritmo YOTT

Algoritmos que abordam a estratégia de travessia em um grafo irão tomar decisões durante o percurso. Os algoritmos de travessia propostos na literatura [Lai et al. 2005, Ferreira and 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. Porém, 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.

As reconvergências geram problemas em vários algoritmos de busca. A Figura 3 exemplifica com dois caminhos. A Figura 3(a) ilustra o caminho percorrido utilizando

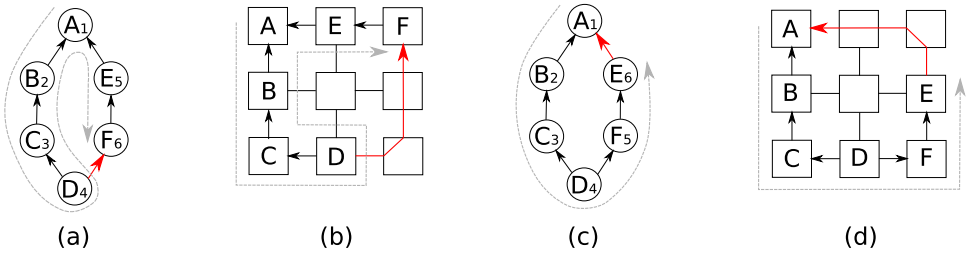

Figura 3. Reconvergência em grafos de fluxo de dados: (a) Percurso em Profundidade; (b) Solução no grafo da arquitetura em profundidade; (c) Percurso em Zigzag; (d) Mapeamento na arquitetura alvo para o percurso em Zigzag.

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(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(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(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.

A Figura 4(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 sequência 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.

A Figura 4(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 4(b) mostra que quando o nodo F será posicionado pela 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

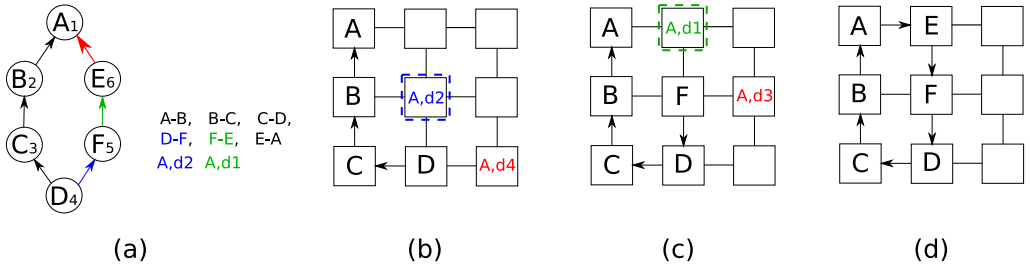

Figura 4. Algoritmo YOTT: (a) Grafo com anotações; (b) Posicionamento de F seguindo as anotações; (c) Posicionamento de E; (d) Solução ótima resultante das anotações do YOTT.

a anotação. A próxima aresta $F \rightarrow E$ será usado novamente a anotação para guiar E na direção de A como ilustrado na Figura 4(c). Finalmente, a Figura 4(d) apresenta o posicionamento e roteamento resultante que é ótimo para o grafo da Figura 4(a).

8. Resultados

O primeiro resultado da dissertação mostrou que o uso do mapeamento com múltiplas travessias em GPU [Canesche et al. 2020] reduziu o tempo de execução para solução exata em 5 ordens de grandeza, executando com um tempo médio de 54.3 microssegundos em comparação com o tempo médio de 72 segundos da solução usando programação linear (ILP) [Walker and et al. 2019]. Ademais, uso das técnicas de ILP para grafos com mais de 25 vértices, enquanto que a múltipla travessia resolveu de forma exata grafos com até 54 nodos.

Tabela 1. Algoritmo Traversal [Canesche et al. 2020]. Pior caso e médias geométricas para filas nas topologias em malha e 1hop. (menor é melhor)

	Pior caso					
	Depth	Zigzag	VPR	(D,1000)	(Z,1000)	GPU
mesh	18	9	9	14	6	7
1hop	10	5	6	7	3	4
	Média geométrica					
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

Considerando a múltipla travessia [Canesche et al. 2020], a Tabela 1 avalia a qualidade da solução medindo o tamanho máximo das filas de balanceamento para seis estratégias: (a) Depth para o percurso de profundidade, (b) a travessia em ZigZag, (c) a ferramenta VPR [Murray and *et al* 2020] (estado da arte em SA), (d) D1000 é o método Depth, mas com 1000 instâncias executadas, (e) Z1000 é o ZigZag, utilizando 1000 instâncias. A versão paralela em GPU utiliza o método SA com 1000 instâncias executadas simultaneamente. Cada instância foi executada por uma thread. O Z1000 apresentou a melhor solução para ambas topologias, tendo uma média geométrica de 1,7 em relação à topologia malha e 1,3 na topologia 1hop. A ferramenta VPR [Murray and *et al* 2020] gerou uma solução pior do que o Z1000 e a implementação GPU.

Tabela 2. Aceleração (em comparação ao VPR) e média do comprimento da FIFO.

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

Aceleração do tempo de execução normalizado pelo tempo de execução do VPR (maior é melhor).

Redução Filas	1	0.80	0.94	1.66	1.35	1.96	1.49	2.24	1.92

Fator de redução da Filas normalizados em comparação com o VPR BB (bound boxing), maior é melhor.

A Tabela 2 compara a aceleração no tempo de execução e a qualidade da solução dos algoritmos múltipla travessia (YOTO) [Canesche et al. 2020] e YOLT [Canesche et al. 2021] com o VPR [Murray and *et al* 2020]. Os resultados estão normalizados em relação ao VPR *Bound Box*. Podemos observar que as implementações por travessia são até 3 ordens de grandeza mais rápidas ($1848\times$ /$766\times$) que o VPR e preservam a qualidade, medida pela redução do tamanho das filas, também normalizado em relação ao VPR. O YOLT é de 2 a 3 vezes mais lento por executar uma dupla travessia em relação ao YOTO. Porém, melhora a qualidade em 1.6 $\times$ em média. Finalmente, na última coluna mostramos que o Yolt100 é melhor que o Yoto1000, sendo $2\times$ mais rápido e com maior redução das filas de balanceamento. Ou seja, a dupla travessia executa melhor a transferência de localidade do grafo de fluxo de dados, resultando em um melhor posicionamento com uma busca guiada mais inteligente.

O algoritmo de dupla travessia (YOTT) mostrou ser possível melhorar a qualidade sem degradar o tempo de execução. A Tabela 3 apresenta os resultados médios para o conjunto de *benchmarks*[1] 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 reduzir em 15% o uso de filas. Entretanto, o tempo da execução do SA é em média 784x mais lento que o YOTT100.

Tabela 3. Tabela comparativa YOTT versus SA

		Média
Algoritmo	Fifo	Tempo (ms)
YOTT100	0.96	**6.80**
SA10	1.78	91.30
SA100	1.52	619.13
SA1000	**0.83**	5340.87

9. Conclusão

Esta dissertação apresentou contribuições relevantes para o problema de posicionamento em CGRA com mapeamento espacial em pipeline. A maior contribuição foi o desenvolvimento de novos algoritmos de travessia polinomiais que capturando a localidade dos grafos, melhorando o estado da arte. Além disso, novas oportunidades poderão explorar ainda mais estas abordagens gerando qualidade e desempenho.

[1]Dataflow graph benchmarks. https://github.com/canesche/benchmarks

Referências

Canesche, M., Carvalho, W., Reis, L., Oliveira, M., Magalhaes, S., Jamieson, P., Nacif, J. M., and Ferreira, R. (2021). You only traverse twice: A yott placement, routing, and timing approach for cgras. *ACM Trans. on Embedded Comp. Systems (TECS)*.

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 Trans. 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 *International Conf. on Field-Programmable Technology (ICFPT)*. IEEE.

Ferreira, R. and et al. (2007). A polynomial placement algorithm for data driven coarse-grained reconfigurable architectures. In *IEEE Symposium on VLSI*.

Ferreira, R. and *et al* (2013). A run-time graph-based polynomial placement and routing algorithm for virtual fpgas. In *Field programmable Logic and Applications (FPL)*.

Gobieski, G. and et al. (2021). Snafu: an ultra-low-power, energy-minimal cgra-generation framework and architecture. In *Int Symp on Computer Architecture (ISCA)*.

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*.

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

Liu, D. and *et al* (2018). Data-flow graph mapping optimization for cgra with deep reinforcement learning. *IEEE Trans. on CAD of Integrated Circuits and Systems*.

Murray, K. E. and *et al* (2020). Vtr 8: High-performance cad and customizable fpga architecture modelling. *ACM Trans on Reconfigurable Technology and Systems (TRETS)*.

Nowatzki, T. and *et al* (2018). Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign. In *Parallel Architectures and Compilation Tech*.

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*. SBC.

Silva, L. B. D. and *et al* (2019). Ready: A fine-grained multithreading overlay framework for modern cpu-fpga dataflow applications. *ACM Trans on Embedded Comp Systems*.

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 *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE.

Walker, M. and et al. (2019). Generic connectivity-based cgra mapping via integer linear programming. In *Field-Programmable Custom Computing Machines (FCCM)*.

Yin, S. and et al. (2017). Dfgnet: Mapping dataflow graph onto cgra by a deep learning approach. In *IEEE Int Symp on Circuits and Systems (ISCAS)*.

Yoon, J. W. and et al. (2009). A graph drawing based spatial mapping algorithm for coarse-grained reconfigurable architectures. *IEEE Trans on VLSI systems*, 17(11).