Simulated Annealing em Hardware com Múltiplas Threads em Pipeline para Posicionamento em CGRAs*

Jeronimo Costa Penha[1,3], Lucas Braganca da Silva[1], Michael Canesche[2], José Augusto M. Nacif[1], Ricardo Ferreira[1]

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

[2]Universidade Federal de Minas Gerais (UFMG)
Av. Antônio Carlos – 6627 – 31270-901 – Belo Horizonte – MG – Brazil

[3]Centro Federal de Formação Tecnológica de Minas Gerais (CEFET-MG)
Av. Amazonas – 5253 – 30421-169 – Belo Horizonte – MG – Brasil

{lucas.bragança,jnacif,ricardo}@ufv.br

jeronimopenha@gmail.com,mcanesche@ufmg.br

Abstract. *O uso de aceleradores com paralelismo espacial, como os CGRAs, são soluções promissoras em desempenho e eficiência energética. O desempenho dos CGRAs dependem dos compiladores para explorar o paralelismo das aplicações, sendo o mapeamento da aplicação um dos grandes desafios. A primeira etapa deste processo é o posicionamento, cuja eficiência impacta diretamente nos passos seguintes que são o roteamento e o escalonamento. Este trabalho apresenta uma implementação em hardware, usando field-programmable gate arrays (FPGA), para o algoritmo Simulated Annealing (SA) sendo a abordagem mais eficiente para o posicionamento. Os resultados mostram uma aceleração de 7 a 30 vezes em relação ao estado da arte sem sacrificar a qualidade da solução. O algoritmo foi implementado em pipeline com múltiplas threads para esconder a latência, mostrando ser possível realizar uma iteração completa do SA em dois ciclos de relógio em FPGA.*

1. Introdução

O problema de posicionamento e roteamento para circuitos digitais, seja circuitos integrados ou em FPGAs, é NP-completo [Hamzeh et al. 2012]. Para circuitos reconfiguráveis de grão grosso (CGRAs), o desafio é ainda maior pois envolve o escalonamento. Em FPGAs, os elementos lógicos são interligados com o objetivo de reduzir o número e comprimento máximo dos fios. No CGRA é necessário o equilíbrio de todos os caminhos para que o fluxo de dados seja processado corretamente, com o escalonamento correto. Este requisito pode gerar a necessidade do uso de recursos de balanceamento como Filas e *buffers* [Nowatzki et al. 2018]. Estes recursos têm alto custo de hardware e podem dobrar o custo total da solução. Uma forma de mitigar este problema é a geração de um posicionamento de qualidade. Outro ponto importante, é que ao restringir o número máximo de filas, a complexidade do mapeamento aumenta significativamente [Nowatzki et al. 2018].

*FAPEMIG, CNPq (Grants #313312/2020-6 and #440087/2020-1 – CNPq/AWS 032/2019) Nvidia, Funarbe.

Várias abordagens podem ser aplicadas, nas quais as etapas de posicionamento, roteamento e escalonamento podem ser realizadas independentemente ou em conjunto.

Soluções com programação inteira [Walker and Anderson 2019] ou resolvedores SAT [Donovick et al. 2019] geram a solução exata mas não escalam bem, por serem limitadas a grafos de fluxo de dados com 30 vértices ou operadores. Outras abordagens usam arquiteturas com poucos elementos de processamento e *modular scheduling* [Hamzeh et al. 2012], porém o desempenho sofre impacto da multiplexação no tempo. Recentemente, abordagens híbridas [Nowatzki et al. 2018] com a mesclagem de programação inteira e soluções gulosas, técnicas de aprendizado por reforço [Liu et al. 2018] e redes neurais com grafos (GNN) [Li et al. 2022] foram propostas. Entretanto, todas as técnicas apresentam limitações de escalabilidade e tempo de compilação, o que eleva da ordem de segundos a minutos, para grafos com poucas dezenas de operações. As soluções mais eficientes envolvem a implementação das etapas separadamente, onde a etapa de posicionamento é implementada com *Simulated Annealing* (SA) [Weng et al. 2022, Carvalho et al. 2020, Murray et al. 2020].

As abordagens com o SA geram resultados com qualidade, mas requerem um tempo de execução maior para uma melhor exploração do espaço de soluções. Este artigo apresenta uma implementação em hardware para o algoritmo SA, desenvolvida em FPGA, que reduz de 7 a 30 vezes o tempo do posicionamento em comparação ao estado da arte [Carvalho et al. 2020, Murray et al. 2020]. As principais contribuições são: (a) modelagem em *pipeline* do algoritmo SA; (b) Execução de uma iteração completa do algoritmo com uma operação de troca (*swap*) em apenas dois ciclos de relógio; (c) Uso de múltiplas *threads* para esconder a latência do *pipeline*; (d) Um simulador e um gerador de código para desenvolvimento de implementações de SA em hardware; (e) Execução com múltiplas unidades de SA que pode reduzir o tempo de execução de 70 a 300 vezes em comparação com a solução em software.

Este artigo está estruturado da seguinte forma. Na seção 2 são apresentadas as implementações paralelas de SA. A seção 3 ilustra o algoritmo básico de SA para posicionamento em CGRAs. A seção 4 detalha a implementação em hardware para o SA proposta neste trabalho. Finalmente, as seções 5 e 6 mostram a avaliação da solução proposta e as principais conclusões e direções futuras.

2. Trabalhos Relacionados

O problema de posicionamento, abordado neste trabalho, consiste em mapear um grafo em uma estrutura bidimensional semelhante a uma malha 2D. A entrada é um grafo de fluxo de dados que implementa uma aplicação que explora o paralelismo temporal e espacial. A arquitetura alvo é uma malha bidimensional de elementos de processamento que irão receber os vértices ou nós do grafo. O desafio é o posicionamento dos nós adjacentes em elementos de processamento vizinhos.

O algoritmo SA para posicionamento começa com uma solução aleatória [Murray et al. 2020] e realiza dezenas de milhares de operações de troca de posição (*swaps*). Com o intuito de escapar dos mínimos locais, é utilizada a técnica de resfriamento do SA que permite aceitar soluções que pioram a qualidade na busca de uma solução mínima global. A Figura 1(a) mostra um grafo e Figura 1(b) apresenta um posicionamento. Os nós e e d são adjacentes ao nó a no grafo. Contudo, não foram

posicionados em células próximas na arquitetura (Figura 1(b)). Ao trocar a e b de células, a ficará mais perto de d e e. No entanto, se ao mesmo tempo trocar d de posição, a aresta $a \rightarrow d$ ficará longa na arquitetura novamente.

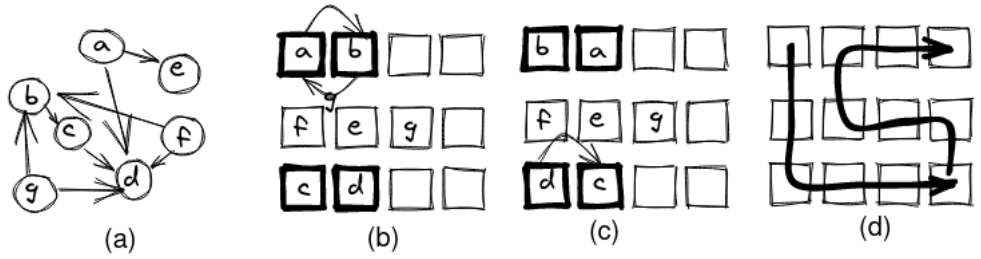

Figura 1. (a) Grafo; Arquitetura: (b) $a \leftrightarrow b$; (c) $c \leftrightarrow d$; (d) Atualização em $O(N)$

Uma implementação em hardware foi proposta em [Wrighton and DeHon 2003] com um arranjo sistólico no formato do CGRA. Cada elemento de processamento armazena localmente um vértice do grafo e uma estrutura de dados com a informação sobre os nós adjacentes e suas posições. O arranjo permite que sejam realizadas trocas em paralelo como ilustra a Figura 1(b-c). Entretanto, uma troca pode afetar o custo de outra, se efetuada como ilustra a Figura 1(c) para os nós c e d, pode alterar os custos das conexões com a e b que não foram atualizadas. Portanto, a solução distribuída permite trocas em paralelo, porém é necessária a atualização após cada troca que envolve uma propagação dos novos valores por todos os elementos do CGRA, como ilustra a Figura 1(d), com um custo $O(N)$, onde N é o número de elementos. A implementação foi apenas simulada com um custo estimado de 150 ciclos para realizar até 4 trocas. Considerando uma frequência de relógio de 300 Mhz dos FPGAs atuais, isto equivale a 500 ns. Entretanto, como iremos mostrar, uma implementação de SA [Carvalho et al. 2020] em um processador com 16 núcleos requer apenas 50ns, ou seja, é um grande desafio criar uma nova solução em hardware mais rápida que as soluções atuais em software. Os resultados mostraram aceleração na execução mas com perda de qualidade em comparação com o VPR [Murray et al. 2020] na opção *fast*. Outra crítica é o consumo de recursos, apesar de não estimar, a proposta do arranjo sistólico é para um FPGA, onde o arranjo seria de 10-100$\times$ maior que o circuito de FPGA que está sendo posicionado.

[Wang and Lemieux 2011] apresenta uma implementação com múltiplos núcleos, onde a versão com 64 núcleos é 20$\times$ mais rápida que o VPR com a opção fast [Murray et al. 2020]. A estratégia é fazer as trocas em paralelo. Uma versão em GPU apresentada em [Fobel et al. 2014] é 19$\times$ mais rápida que o VPR. Uma implementação com programação dinâmica e GPU é apresentada em [Dhar and Pan 2018] com um ganho de 10$\times$ em relação a uma versão com múltiplos núcleos. Outra abordagem, apresentada por [Kong et al. 2020], explora características do circuito para a geração de um posicionamento 20$\times$ mais rápido que o VPR. Entretanto, as soluções não melhoram a qualidade da solução e a maioria até gera perdas de qualidade para obter ganhos de desempenho.

A maioria das implementações paralelas de posicionamento tratam o problema para FPGA, mas o nosso foco neste trabalho são os CGRAs. Apesar dos grafos serem bem menores nos CGRAs, o problema é mais complexo pois envolve o escalonamento e balanceamento dos caminhos. Como já mencionado, várias

técnicas escalam apenas para grafos com até 30 vértices utilizando programação inteira [Walker and Anderson 2019], resolvedores SAT [Donovick et al. 2019] ou abordagens híbridas [Nowatzki et al. 2018]. Recentemente, técnicas de inteligência artificial como o aprendizado por reforço [Liu et al. 2018] e redes neurais com grafos [Li et al. 2022] foram propostas. Estes trabalhos apresentam tempos de execução elevados e foram avaliados apenas em arquiteturas com poucos elementos de processamento e multiplexação no tempo.

Finalmente, podemos concluir que várias abordagens foram propostas para o posicionamento, sendo que o SA continua a apresentar, por décadas [Mulpuri and Hauck 2001, Wang and Lemieux 2011], o melhor custo benefício entre tempo de execução e qualidade. Entretanto, reduzir o tempo ainda é um desafio, com técnicas paralelas acelerando 10-20 $\times$ em múltiplos núcleos e GPUs. Este trabalho apresenta um avanço no estado da arte com uma implementação em hardware do SA que é até 30 vezes mais rápida que o VPR [Comodi and Michalak 2022], ao realizar uma iteração com troca em apenas 2 ciclos de relógio e gerar resultados que melhoram a qualidade, pois reduzem a quantidade de recursos que o escalonamento do CGRA utiliza. Ademais, o acelerador usa poucos recursos do FPGA e pode ser replicado, executando várias cópias para gerar um ganho de desempenho de até 300 vezes em comparação ao VPR.

3. Algoritmo *Simulated Annealing*

A Figura 2(a) apresenta um pseudocódigo do algoritmo SA para o problema de posicionamento. Pode-se observar que existe uma sequência de ações com dependência de dados. Primeiro, duas células C_a e C_b são escolhidas aleatoriamente para serem trocadas. Depois é necessário identificar quais nós estão posicionados nas mesmas. No exemplo são os nós a e b. Para tanto é necessária uma estrutura para o mapeamento $célula \rightarrow nó$, denominada $celula2no$ ou C_2N. Depois é preciso consultar o grafo para descobrir quais são os nós adjacentes ou os vizinhos de a e os vizinhos de b. Suponha que sejam os conjuntos de nós V_a e V_b. Esta informação está armazenada na estrutura $Vizinhos$, que armazena o grafo. O próximo passo é determinar as células que contém cada um dos nós (ou vizinhos) em V_a e V_b, denominados pelos conjuntos de células VC_a e VC_b. Para este passo é necessário a estrutura que mapeia os nós em células ($Nó \rightarrow Celula$ ou N_2C). Finalmente pode-se calcular as distâncias de todos os elementos de VC_a e VC_b para a e b posicionados nas células C_a e C_b e vice-versa, respectivamente.

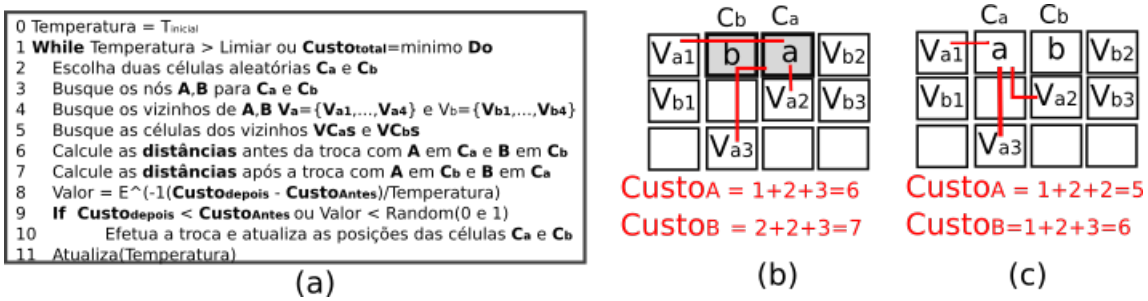

Figura 2. (a) Pseudo-Código do SA; (b) Antes $C_a \leftrightarrow C_b$; (c) Depois da troca.

A Figura 2(b) apresenta em destaque as distâncias dos vizinhos de a. Observa-se que V_{a1} está posicionado em uma célula à esquerda com distância 2 de a. Já V_{a2} e V_{a3}

estão, respectivamente, as distâncias 1 e 3 de a. Assim, o custo das arestas de a é 6. Para o nó b te-se uma distância total 7. Estes são os custos antes da troca entre os nós selecionados. Se a troca for efetuada, como ilustrado na Figura 2(c), o nó a ficará em uma célula mais próxima dos seus vizinhos e o mesmo ocorre para b. Neste caso a troca é aceita. No algoritmo de SA, mesmo que o custo não seja melhor, ela pode ser aceita de acordo com uma probabilidade determinada pela $temperatura$. Este recurso evita que a solução fique presa em mínimos locais. A temperatura começa com um valor mais alto e vai sendo reduzida com o número de iterações o que reduz a probabilidade de trocas ditas ”ruins”sejam efetuadas.

Devido à cadeia de dependência entre os passos do algoritmo, têm-se vários desafios para a paralelização. Este trabalho apresenta uma proposta em pipeline para a exploração do paralelismo temporal e, em cada estágio, são exploradas as tarefas que possam ser executadas em paralelismo espacial. Para esconder a latência gerada pelas dependências, várias cópias do algoritmo são executadas em pipeline. A execução de múltiplas cópias é importante para melhorar a qualidade da solução como mostrado em [Carvalho et al. 2020]. Finalmente, das estruturas apresentadas, apenas N_2C e C_2N devem ser atualizadas para a e b, caso seja efetivada a troca. A estrutura $Vizinhos$ não muda. A próxima seção apresenta o detalhamento da solução ilustrando como ter acesso paralelo com múltiplas *threads* e ao mesmo tempo realizar a atualização com sobreposição de tarefas.

4. *Simulated Annealing* em Pipeline

A Figura 3 apresenta um fluxo resumido dos principais estágios da versão pipeline do SA. Após a escolha de duas células, no próximo estágio, uma memória com o mapeamento célula para os nós (C_2N) é consultada para determinar quais são os nós que estão nas células C_a e C_b, denominados a e b. As linhas que têm uma dependência sequencial no pseudocódigo da Figura 2(a) são mapeadas em estágios sequenciais no pipeline da Figura 3. Assumimos que cada memória é lida em um ciclo de relógio.

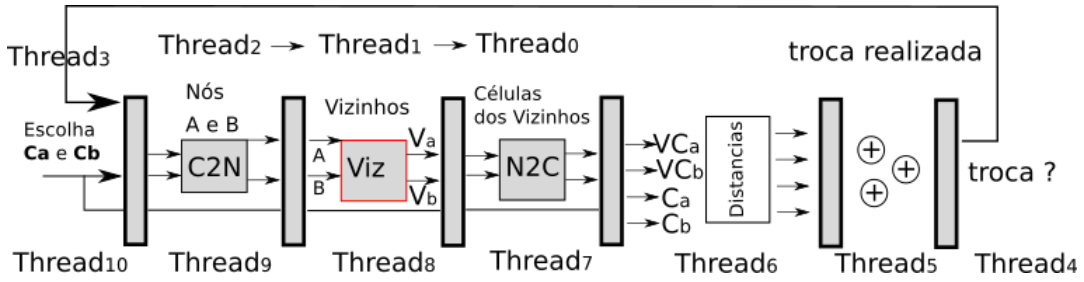

Figura 3. Pipeline simplificado com múltiplas *threads*.

Para esconder a latência das dependências de dados, várias *threads*, com a execução de uma cópia do SA em cada uma, percorrem o pipeline. Assim, a cada ciclo, é possível realizar uma iteração completa incluindo uma troca. Por exemplo, enquanto a $thread_4$ está no último estágio avaliando a troca, a $thread_5$ executa a redução de soma, a $thread_6$ calcula as distâncias entre as células, a $thread_7$ lê a células com a posição dos vizinhos, etc. Outra vantagem da solução é que as estruturas de dados estão distribuídas

ao longo do pipeline. Até mesmo a redução de soma para calcular as distâncias é incluída no pipeline.

As próximas subseções detalham a implementação do SA em hardware.

4.1. Memórias

A maior parte dos dados está armazenada em memórias distribuídas dentro do FPGA. Estas memórias podem ser implementadas usando módulos BRAM ou através da personalização de elementos lógicos para atuarem como memórias. A maioria dos FPGAs oferecem estes recursos com memórias que permitem 2 (ou mais) leituras e uma escrita por ciclo. Após ser lido em um estágio, o dado é repassado para o seguinte e o processamento continua. Dessa forma, o SA gera uma cadeia de ponteiros acessados em pipeline.

4.1.1. Memória de armazenando o Grafo

O grafo é armazenado com a estrutura de lista de adjacência, ou seja, para cada nó é armazenada a lista de nós adjacentes ou vizinhos. Foi escolhido o termo vizinho, mas é importante fazer a distinção em os termos nó e a célula. O termo nó se refere ao grafo e o termo célula indica onde está posicionado o nó na arquitetura. Como o grafo não é alterado, a memória que armazena as listas de adjacências é a mesma para todas as *threads* e pode ser compartilhada. Só é carregada no início e não precisa ser atualizada.

Para validar a implementação, serão utilizados grafos com 4 vizinhos, ou nós adjacentes, dois de entrada e dois de saída. Todos os nós terão no máximo 4 vizinhos com o objetivo de preservar a regularidade do pipeline, mesmo que seja um vizinho vazio. O código é parametrizado e pode ser modificado para trabalhar com mais nós adjacentes, ou seja, um *fan-in* ou *fan-out* maior. Entretanto, qualquer grafo pode ser decomposto em nós com *fan-in* (entradas) e *fan-out* (saídas) limitados a 2.

4.1.2. Quantidade de Módulos de Memória

Para o exemplo, será necessária a avaliação de 8 vizinhos, pois são avaliados dois nós com 4 vizinhos cada. O desafio é: como realizar 8 leituras em paralelo? O FPGA permite, ao menos, a realização de duas leituras por módulo de memória. Uma solução é ter vários estágios com cópias da memória para a leitura dos vizinhos, com a leitura de 2 por estágio. Outra solução é ter várias cópias da memória de vizinhos em um único estágio e realizar todas as leituras simultaneamente com várias memórias. Como mencionado, as várias cópias são necessárias para a leitura dos 8 vizinhos em pipeline. O gerador de código pode ser configurado para ambas as opções: um estágio com 4 memórias ou 4 estágios com uma cópia da memória por estágio.

4.1.3. Dados privados de cada *Thread*

As memórias que realizam mapeamento entre o nó e a célula (N_2C e C_2N) possuem áreas reservadas para cada *thread*. A parte mais significativa do endereço é indexada

com o índice da *thread* e a menos significativa com o índice do nó ou da célula. Portanto, o aumento no número de *threads* gera impacto direto no tamanho destas memórias. Ademais, estas memórias também são distribuída e duplicada em vários estágios. A mais crítica é a N_2C, pois, para cada nó adjacente das listas de vizinhos, é necessário determinar em qual célula cada nó vizinho está posicionado e usar esta informação para calcular as distâncias. Ou seja, são necessários 8 acessos a 4 módulos de memória replicados em paralelo.

4.1.4. Atualização em dois ciclos

O ponto mais importante para ser destacado é a atualização das memórias C_2N e N_2C. Caso a troca seja efetivada, a posição de a será C_b e a posição de b será C_a. Então $N_2C(b) = C_a$ e $N_2C(a) = C_b$. Portanto, é necessário realizar duas escritas, mas só é possível realizar uma por ciclo em cada memória. Este problema é resolvido com o processamento de uma *thread* a cada dois ciclos. Portanto, no lugar de $Thread_6 \rightarrow Thread_5 \rightarrow \ldots$ teremos $Thread_6 \rightarrow -- \rightarrow Thread_4 \rightarrow -- \rightarrow \ldots$, ou seja, as *threads* são inseridas a cada 2 ciclos. Em um ciclo a nova posição do nó b é atualizada com $N_2C(b) \leftarrow C_a$ e no estágio seguinte, no próximo ciclo, a posição do nó a é atualizada com $N_2C(a) \leftarrow C_b$. O mesmo é válido para marcar o novo conteúdo das células com as escritas $C_2N(C_b) \leftarrow a$ e $C_2N(C_a) \leftarrow b$.

Além disso, os primeiros estágios são visitados duas vezes por cada *thread* que percorre o pipeline. Na primeira visita, a *thread* passa lendo os dados, como por exemplo a $thread_9$ que faz a leitura dos nós que estão nas células C_a e C_b no segundo estágio do pipeline ilustrado nas Figuras 3 e 4. Ao mesmo tempo, este estágio está sendo visitado pela $thread_2$ que irá atualizar o novo conteúdo da C_a e C_b da sua cópia, caso a troca tenha sido efetivada para $thread_2$. Como estes dados são privados para cada *thread* enquanto a $thread_9$ efetua a leitura, a $thread_2$ faz a escrita em regiões diferentes da memória.

4.2. Propagação dos dados

Durante a execução, os dados das *threads* se propagam ao longo do pipeline. Por exemplo, C_a e C_b são propagados por todos os estágios até o cálculo da distância onde é consumido pela última vez. Outros dados são propagados apenas de um estágio para o outro como os nós a e b. Entretanto, para efetivar a atualização da troca, a, b, C_a e C_b teriam que percorrer todo o pipeline e depois retornar aos estágios de atualização de N_2C e C_2N. Para reduzir o número de sinais ao longo do pipeline, a implementação usa uma fila local que armazena estes dados no segundo estágio para posteriormente propagá-lo na fase de atualização. A fila tem o comprimento da latência do pipeline. Portanto, apenas um bit deve retornar do último estágio para o primeiro para informar se teve ou não troca e assim na fase de escrita, a *thread* fará a atualização.

A Figura 4 mostra em destaque os dois pontos importantes. As *threads* entram a cada dois ciclos para possibilitar a atualização da troca, pois só é permitida uma escrita por ciclo nas memórias. Primeiro, a célula a é atualizada na posição C_b, se ocorreu a troca. Os valores de C_a e b passam para o próximo estágio. No próximo ciclo, quando a $thread_1$ estará no próximo estágio, a atualização de C_b e a é realizada. A fila armazena as células e nós no momento que a *thread* passa para leitura. No exemplo da Figura 4,

a $thread_6$ que está passando pelo estágio e ao mesmo tempo a $thread_1$ está na fase de escrita, semelhante ao estágio $writeback$ dos processadores RISC.

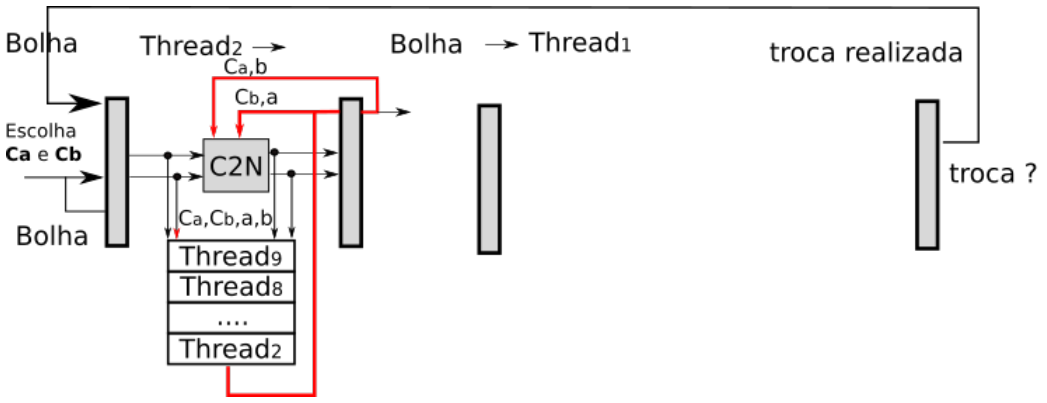

Figura 4. Como atualizar com múltiplas *threads* e redução de sinais.

5. Resultados

Esta seção está estruturada da seguinte forma: primeiro é apresentada a metodologia de desenvolvimento da implementação nas seções 5.1 e 5.2. A seção 5.3 mostra o consumo de recursos da versão em hardware do SA para ser implementado em um FPGA. A seção 5.4 apresenta uma avaliação de desempenho em comparação com VPR [Murray et al. 2020] e um SA com múltiplos núcleos [Carvalho et al. 2020].

5.1. Simulador em Python

Para flexibilizar o desenvolvimento da implementação, foi desenvolvido um simulador em Python que permite avaliar diversas configurações de troca e número de estágios. Outro trabalho [Wrighton and DeHon 2003] com implementação em hardware com arranjo sistólico também desenvolveu um simulador. O simulador pode ser usado para depurar a execução mostrando o conteúdo das memórias, a evolução da redução do custo das soluções por *thread* e os posicionamentos parciais. Outro detalhe importante é que o simulador simplifica e valida o desenvolvimento de um gerador de código descrito na próxima seção.

5.2. Gerador de código Parametrizado

A implementação do gerador de código foi feita em linguagem Python com o auxílio da biblioteca Veriloggen [Takamaeda 2015]. Esta biblioteca é especializada na geração de código Verilog a partir de funções pré-definidas em Python. O gerador é parametrizável com o tamanho da arquitetura alvo. Como o pipeline tem uma estrutura regular, o uso do gerador permite a exploração futura de várias possibilidades de projetos variando a profundidade do pipeline e/ou a complexidade dos estágios, a distribuição dos módulos de memória ao longo dos estágios entre outros aspectos que podem ser investigados.

5.3. Recursos em FPGA

O código Verilog gerado foi sintetizado com Vivado 2020.2 para o FPGA Xilinx VU9P que possui aproximadamente 2,5 milhões LUTs, 6800 DSPs e 2160 bancos de

Tabela 1. FPGA Xilinx VU9P e utilização de recursos

Tam	LUT(%)	REG(%)	Lut as RAM	Bram(%)	MEM Kb
4x4	1454 (0,1)	1535 (0,1)	280 (0,1)	0 (0,0)	4
10c	14089 (1,6)	14786 (0,8)	2800 (0,5)	0 (0,0)	45
8x8	3300 (0,3)	1705 (0,1)	784 (0,1)	0 (0,0)	13
10c	32469 (3,7)	16486 (0,8)	7840 (1,4)	0 (0,0)	125
9x9	8688 (0,9)	1895 (0,1)	564 (0,1)	4 (0,2)	137
10c	86769 (9,8)	18426 (0,9)	5640 (1,0)	40 (2,2)	1370
10x10	17980 (1,8)	2147 (0,1)	564 (0,1)	4 (0,2)	137
10c	177609 (20,0)	20636 (1,1)	5640 (1,0)	40 (2,2)	1370
12x12	36390 (3,6)	1823 (0,1)	1240 (0,2)	4 (0,2)	148
10c	363449 (40,9)	17666 (0,9)	12400 (2,2)	40 (2,2)	1478

memória BRAM com capacidade total de 7,9MB de memória no FPGA [Xilinx 2020]. A tabela 1 apresenta o total de recursos utilizados para diversos tamanhos de arquitetura.

Para a validação do projeto, foram gerados circuitos de 1 e 10 cópias (linha 10c) com arquiteturas 4x4, 8x8, 9x9, 10x10 e 12x12. Todos os circuitos atingiram a frequência máxima da plataforma que é de 250MHz. Observa-se ainda na tabela 1 que as diferenças entre o consumo de recursos para os circuitos de 1 e 10 cópias foram aproximadamente de 10 vezes, o que mostra a escalabilidade da solução.

Na última coluna da Tabela 1 encontram-se os valores de memória alocados pelo sintetizador para cada instância. Pode-se avaliar o custo e a escalabilidade da solução com o consumo de memória. A necessidade de RAM depende diretamente da quantidade de *threads* que poderão ser executadas e do tamanho da arquitetura. Como número da *thread* determina o endereço mais significativo, temos três patamares de consumo que escalam com potência de 2 para arquitetura até 4x4, até 8x8 de 9x9 até 15x15. Por exemplo, para 9x9 e 10x10 temos 81 e 100 células que é menor 128 (potência de 2 mais próxima). Conforme os circuitos exigem memórias com maior profundidade, blocos de BRAM começaram a ser utilizados pelo sintetizador. Como cada bloco possui 32Kb de capacidade de armazenamento, os blocos ficarão com posições inutilizadas, uma vez que a demanda é bem menor. Mas não tem impacto no uso de recursos por o consumo em blocos é baixo, no máximo fica em 2% da capacidade total do FPGA.

Por fim, o único recurso demandado em maior escala são as LUTs, pois a implementação da medida de distância foi realizada com lógica combinacional. Este passo pode ser futuramente substituído por DSP e será reduzido significativamente. O gerador é modular e pode ser facilmente melhorado.

5.4. Avaliação do Desempenho

A ferramenta VPR [Murray et al. 2020] representa o estado da arte nas duas últimas décadas para SA, sendo atualmente usada em uma parceria da Google com a Universidade de Toronto no projeto Antmicro [Comodi and Michalak 2022] para desenvolvimento de ferramentas de código aberto para projeto de circuitos integrados e FPGAs.

A Tabela 2 apresenta o tempo de execução do posicionamento para um conjunto

Tabela 2. (Tempo para posicionamento de um conjunto de aplicações com VPR opção *fast*, SA com 16 threads no processador AMD Ryzen 7 3700X, considerando a execução de 1000 instâncias e seleção da melhor solução.

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

de grafos de fluxo de dados para aplicações de processamento de sinais e aprendizado de máquina. Às duas primeiras colunas identificam a aplicação e o tamanho em nós do grafo de fluxo de dados. A coluna seguinte apresenta quantas iterações foram realizadas considerando 1.000 execuções (ou 1.000 threads) de SA. O uso de 1.000 execuções melhora significativamente o resultado. A coluna T_{16th} mostra o tempo de execução para a implementação proposta em [Carvalho et al. 2020] considerando 16 threads executando no processador AMD Ryzen 7 3700X de 4.4 GHz com 40 M de cache L3 e 64 GB RAM. A coluna T_{Vpr} mostra o tempo de execução para 1.000 cópias do VPR com a opção *fast* ativada. Pode-se observar que a implementação T_{16th} [Carvalho et al. 2020] é em média 4,3 vezes mais rápida que o VPR na opção *fast*. As duas colunas rotuladas como fila mostram que a qualidade da solução medindo o tamanho da maior fila de balanceamento da versão T_{16th} em comparação com o VPR, destacando que o posicionamento com 16 Threads gerou um escalonamento de qualidade superior ao VPR. Quanto menor o tamanho da fila, melhor. Resumindo, estes resultados mostram que o SA resolve o problema de posicionamento com qualidade e que as implementações paralelas com múltiplos núcleos aceleram o VPR.

A síntese mostrou que o FPGA executa com no mínimo uma frequência de relógio de 250 MHz. Portanto, a implementação de SA em hardware executa em apenas 8 nanosegundos, uma iteração completa com (ou sem) troca. Comparando com a versão T_{16th}, cujo tempo médio da iteração é 59,3 nano segundos, a implementação em hardware proposta neste trabalho é 7,4 $\times$ e 31,5 $\times$ mais rápida que a versão paralela com múltiplos núcleos e VPR, respectivamente. Na comparação, consideramos a média geométrica dos tempos de execução, com melhoria da qualidade em relação ao VPR e preservando a qualidade da versão de 16 threads. Quando são utilizadas 10 unidades de SA, a implementação proposta é 315 vezes mais rápida que a ferramenta VPR, sem consumir muitos recursos do FPGA.

6. Conclusão

Este trabalho apresentou uma implementação em hardware para o algoritmo *Simulated Annealing* aplicado ao problema de posicionamento de aceleradores de grafos de fluxo de dados em CGRAs. A implementação fez uso de pipeline com múltiplas threads para paralelizar a execução. Comparando com a ferramenta VPR [Murray et al. 2020], a versão em hardware foi, em média, de 30 à 300 vezes mais rápida e além de melhorar a qualidade da solução ao reduzir o número de filas necessárias para o balanceamento. A solução paralela usa várias memórias e uma solução de sobreposição de escrita para maximizar o desempenho, executando cada iteração do SA em apenas 2 ciclos de relógio. Trabalhos futuros irão avaliar a integração de um algoritmo de roteamento em hardware.

Referências

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

Comodi, A. and Michalak, T. (2022). Fpga format. https://opensource.googleblog.com /2022/02/FPGAInterchangeformattoenableinteroperableFPGAtooling.html.

Dhar, S. and Pan, D. Z. (2018). Gdp: Gpu accelerated detailed placement. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. 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.

Fobel, C., Grewal, G., and Stacey, D. (2014). A scalable, serially-equivalent, high-quality parallel placement methodology suitable for modern multicore and gpu architectures. In *International Conference on Field Programmable Logic and Applications (FPL)*.

Hamzeh, M., Shrivastava, A., and Vrudhula, S. (2012). Epimap: Using epimorphism to map applications on cgras. In *IEEE Design Automation Conference*.

Kong, H., Feng, L., Deng, C., Yuan, B., and Hu, J. (2020). How much does regularity help fpga placement? In *Int Conf on Field-Programmable Technology (ICFPT)*.

Li, Z., Wu, D., Wijerathne, D., and Mitra, T. (2022). Lisa: Graph neural network based portable mapping on spatial accelerators. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 444–459. IEEE.

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 Trans on Computer-Aided Design of Integrated Circuits and Systems*, 38(12).

Mulpuri, C. and Hauck, S. (2001). Runtime and quality tradeoffs in fpga placement and routing. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 29–36.

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.

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

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

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

Wang, C. C. and Lemieux, G. G. (2011). Scalable and deterministic timing-driven parallel placement for fpgas. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 153–162.

Weng, J., Liu, S., Kupsh, D., and Nowatzki, T. (2022). Unifying spatial accelerator compilation with idiomatic and modular transformations. *IEEE Micro*, (01):1–12.

Wrighton, M. G. and DeHon, A. M. (2003). Hardware-assisted simulated annealing with application for fast fpga placement. In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 33–42.

Xilinx, D. (2020). Ultrascale architecture and product data sheet: Overview. *Product Specification, Nov*.