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

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

[1]Universidade Federal de Viçosa (UFV)
[2]Universidade Federal de Minas Gerais (UFMG)
[3] Centro Federal de Formação Tecnológica de Minas Gerais (CEFET-MG)
jeronimopenha@gmail.com,ricardo@ufv.br

Resumo. *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). Os resultados mostram uma aceleração de 7 a 30 vezes em relação ao estado da arte sem sacrificar a qualidade da solução, podendo ser de 70 a 300 vezes mais rápido com o uso de múltiplas unidades de posicionamento. O algoritmo foi implementado em pipeline com múltiplas threads para esconder a latência, onde uma iteração completa do SA executa em apenas dois ciclos de relógio do FPGA.*

1. Introdução

O problema de posicionamento e roteamento para circuitos digitais, sejam circuitos integrados ou em FPGAs, é NP-completo [Ferreira et al. 2013]. Para circuitos reconfiguráveis de grão grosso (CGRAs), o desafio é maior pois envolve 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 usando filas de balanceamento [Nowatzki et al. 2018]. Estes são recursos podem dobrar o custo total da solução. Uma forma de mitigar este problema melhorar a qualidade do posicionamento. Outro ponto importante, ao se restringir o número máximo de filas, é que a complexidade do mapeamento aumenta significativamente [Nowatzki et al. 2018].

O uso de programação inteira [Walker and Anderson 2019] ou resolvedores SAT [Donovick et al. 2019] geram soluções exatas porém sem boa escalabilidade, por serem limitadas a grafos de fluxo com 30 vértices ou operadores. Outras abordagens usam arquiteturas com poucos elementos de processamento e *modulo scheduling* [Ferreira et al. 2013], porém o desempenho sofre impacto da multiplexação no tempo. Recentemente, abordagens híbridas [Nowatzki et al. 2018] com a mesclagem

*Financiamento: FAPEMIG APQ–01203-18, CNPq (#313312/2020-6 and #440087/2020-1 – CNPq/AWS 032/2019) NVIDIA, Funarbe. O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001

de programação inteira e soluções gulosas, algoritmos genéticos [Da Silva et al. 2006], 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 para grafos com poucas dezenas de operações. As soluções mais eficientes envolvem a implementação das etapas separadas, 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, desenvolvido 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 explica 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 ou próximos.

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.

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 nó 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 ilustrado a Figura 1(c) para os nós c e d, pode alterar os custos

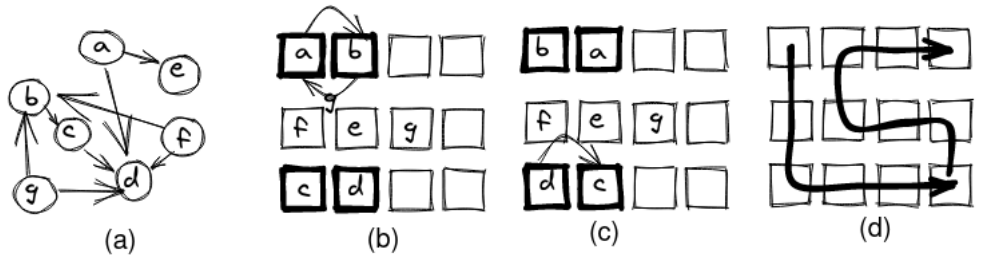

Figura 1. (a) Grafo; Arquitetura: (b) $a \leftrightarrow b$; (c) $c \leftrightarrow d$; (d) Atualização em $O(N)$

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, 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 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*. Além disso, a solução sistólica não é escalável, o arranjo para implementar a unidade de posicionamento é de 10-100$\times$ maior que o circuito que está sendo mapeado no FPGA. Recentemente, uma solução em hardware foi apresentada em [Vieira et al. 2021] com uma abordagem baseada em algoritmos de travessia. Apesar da aceleração no tempo de execução, a qualidade de solução é inferior as implementações com SA [Carvalho et al. 2020].

Várias soluções com múltiplos núcleos e GPUs mostraram aceleração da ordem de 10-20x em comparação com o VPR. [Wang and Lemieux 2011] apresenta uma implementação com 64 núcleos que é 20$\times$ mais rápida que o VPR com a opção fast [Murray et al. 2020]. 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 paralelas supracitadas não melhoram a qualidade da solução e a maioria até gera perdas de qualidade para obter ganhos de desempenho. Neste trabalho aceleramos a execução em 30 a 300x em comparação com o VPR sem perda de qualidade.

A maioria das implementações paralelas de posicionamento tratam o problema para FPGA, mas o foco deste trabalho são os CGRAs. Apesar dos grafos serem menores nos CGRAs, o problema se torna mais complexo pois envolve o escalonamento e balanceamento dos caminhos. Como já mencionado, várias técnicas escalam apenas para grafos com até 30 vértices [Walker and Anderson 2019, Donovick et al. 2019]. Estes trabalhos apresentam tempos de execução elevados e foram avaliados apenas em arquiteturas com poucos elementos de processamento e multiplexação no tempo.

Dentre as várias abordagens propostas para o posicionamento, podemos desta-

car que o SA apresenta o melhor custo benefício entre tempo de execução e qualidade nos resultados [Mulpuri and Hauck 2001, Wang and Lemieux 2011]. Entretanto, reduzir o tempo de execução sem perder qualidade ainda é um desafio. Este trabalho apresenta um avanço no estado da arte com uma implementação em *hardware* que é de 30 vezes mais rápida que o VPR. Cada iteração do SA executa em apenas 2 ciclos. Ademais, o acelerador usa poucos recursos do FPGA e pode ser replicado, o que permite a execução de várias cópias com o objetivo de obter um ganho de desempenho de até 300 vezes em comparação ao VPR.

3. Algoritmo *Simulated Annealing*

A Figura 2(a) apresenta o pseudocódigo do algoritmo SA para o problema de posicionamento. O código possui uma serie de ações com dependência de dados. Primeiro, duas células C_a e C_b são escolhidas aleatoriamente para serem trocadas. Depois são identificados quais nós estão posicionados nestas células usando a estrutura de mapeamento $célula \rightarrow nó$, denominada $celula2no$ ou C_2N. No exemplo são os nós a e b. O próximo passo consulta o grafo para descobrir quais são os nós adjacentes de a e de b, definidos como os conjuntos de nós V_a e V_b, respectivamente. Em seguida, é necessário determinar em quais células estão os nós (adjacentes) de V_a e V_b, denominados pelos conjuntos de células VC_a e VC_b. Este passo faz uso da uma estrutura que armazena o mapeamento dos nós contidos em cada célula ($Nó \rightarrow Célula$ 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.

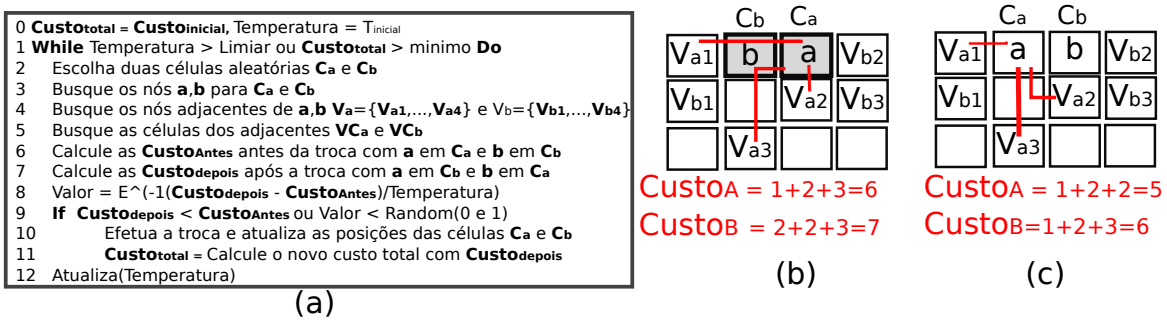

Figura 2. (a) Pseudo-Código do SA; (b) Antes $C_a \leftrightarrow C_b$; (c) Depois da troca.

A Figura 2(b) apresenta em destaque as distâncias dos nós adjacentes 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, a distância total é 7. Estes custos são antes da troca dos nós selecionados. Se a troca for efetuada, como ilustrado na Figura 2(c), o nó a ficará em uma célula mais próxima dos seus adjacentes e o mesmo ocorre para b. Para este caso a troca será aceita. No algoritmo SA, uma troca pode ser aceita mesmo que o custo não seja melhorado com uma probabilidade determinada pela $temperatura$. Este recurso evita que a solução se prenda a mínimos locais. A temperatura é iniciada com um valor mais alto que é reduzido com de acordo com o número de iterações executadas. Este processo reduz progressivamente a probabilidade do aceite das trocas que pioram o custo.

Devido à cadeia de dependências presentes no código do SA, têm-se vários desafios para a paralelização. Este trabalho apresenta uma proposta em *pipeline* para a

exploração do paralelismo temporal. Além disso, em cada estágio do *pipeline*, as tarefas são executadas explorando o paralelismo espacial. Ademais, várias cópias do algoritmo são executadas em *pipeline* para esconder a latência gerada pelas dependências sequenciais. Outro ponto é a execução de múltiplas cópias que é importante para a melhoria da qualidade da solução conforme [Carvalho et al. 2020]. Das estruturas apresentadas, apenas N_2C e C_2N devem ser atualizadas para a e b, em caso de efetivação de troca. A seção 4 detalha o acesso paralelo as memórias que armazenas as estruturas para a execução simultânea de múltiplas *threads*.

4. *Simulated Annealing* em *Pipeline*

A Figura 3 apresenta o 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 a determinação de quais nós estão contidos 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. Assume-se que cada memória é lida em um ciclo de relógio.

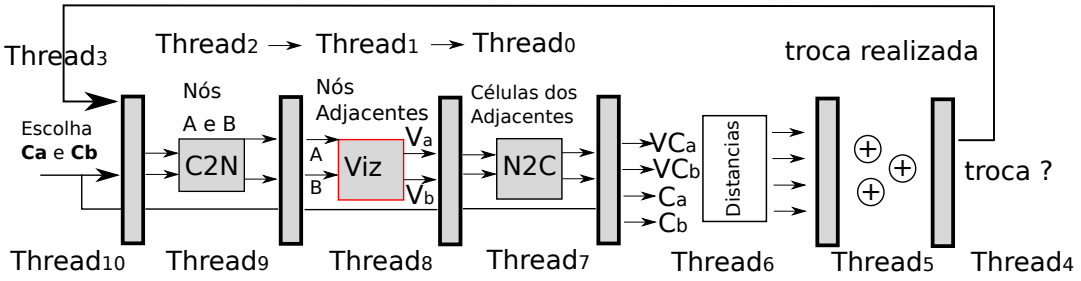

Figura 3. *Pipeline* simplificado com múltiplas *threads*.

A latência das dependências de dados é escondida com a execução de várias *threads*, onde cada uma executa uma cópia do SA. Assim, a cada ciclo é possível realizar uma iteração completa com uma troca. Por exemplo, enquanto a $thread_4$ está no último estágio avaliando se uma troca será feita, 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 adjacentes, etc. Outra vantagem da solução é que as estruturas de dados estão distribuídas ao longo do *pipeline* incluindo o cálculo das distâncias. As próximas subseções detalham a implementação em *hardware*.

4.1. Memórias

A maior parte dos dados está armazenada em memórias distribuídas dentro do FPGA. Estas memórias podem ser implementadas com módulos BRAM ou através dos elementos lógicos. A maioria dos FPGAs oferecem o recurso de utilizar memórias que permitem 2 (ou mais) acessos de leitura e uma escrita por ciclo. Após ser lido em um estágio, o dado é repassado para o estágio seguinte e o processamento continua.

4.1.1. Armazenamento do Grafo

O grafo é armazenado como uma lista de adjacências, ou seja, para cada nó é armazenada uma lista de nós adjacentes. É importante fazer a distinção entre os termos nó e a célula. O termo nó se refere ao grafo e o termo célula indica onde o nó está posicionado na arquitetura. Como o grafo não é alterado, a memória que armazena as listas de adjacências é a mesma para todas as *threads* e é compartilhada. Esta memória é carregada no início da execução.

Para os grafos de fluxo de dados avaliados, os operadores são binários ou unários. Portanto cada possui no máximo dois nós adjacentes. Com relação a saída do nó, qualquer grafo pode ser decomposto em nós com grau máximo de saída 2 com utilização de nós *buffers*. A implementação foi validada com grafos com até 4 nós adjacentes. Entretanto, o código é parametrizado e pode ser modificado para trabalhar com mais nós adjacentes, ou seja, com um *fan-in* ou *fan-out* maior.

4.1.2. Quantidade de Módulos de Memória

Para o exemplo, será necessária a avaliação de 8 adjacências, pois são avaliados dois nós com 4 nós adjacentes 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 adjacentes, com a leitura de 2 por estágio. Outra solução é ter várias cópias da memória de adjacentes em um único estágio e realizar todas as leituras simultaneamente em várias memórias. 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*

Cada *thread* possui uma área reservada nos módulos de memória N_2C e C_2N (Seção 3). Para evitar que haja sobreposição, a parte mais significativa do endereço é o número da *thread* concatenado com o número da célula ou nó selecionado como ilustrado na Figura 4, onde as *threads* 0,1,2 e 3 fazem acesso as posições 2, 0, 99 e 1 da sua região, respectivamente. Portanto, o tamanho das memórias depende da quantidade de *threads* e do número máximo de células (tamanho da arquitetura). Ademais, para garantir acesso paralelos aos dados, foi necessário distribuir e duplicar as memórias em vários estágios do *pipeline*, com mencionado na seção 4.1.2.

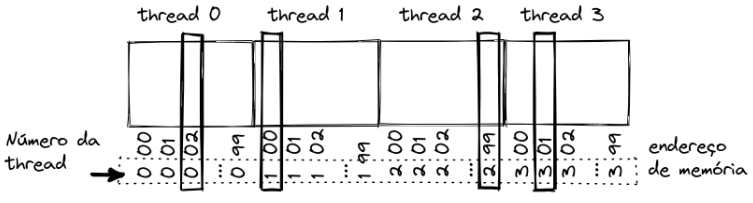

Figura 4. Uso das memórias com múltiplas *threads*.

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, são necessárias duas escritas, porém apenas uma é possível a cada ciclo em cada memória. Este problema é resolvido com o processamento de uma *thread* a cada dois ciclos. Assim, no lugar de $Thread_6 \rightarrow Thread_5 \rightarrow \ldots$ teremos $Thread_6 \rightarrow -- \rightarrow Thread_5 \rightarrow -- \rightarrow \ldots$, 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 a marcação do 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 5. 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 a efetivação da troca, a, b, C_a e C_b seria necessário que percorressem todo o *pipeline* e retornassem aos estágios de atualização de N_2C e C_2N. Para reduzir o número de sinais ao longo do circuito, 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 retorna do último para o primeiro estágio informando se a troca deve ser realizada ou não.

A Figura 5 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 $thread_2$ 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 5, a $thread_9$ que está passando pelo estágio e ao mesmo tempo a $thread_2$ está na fase de escrita, semelhante ao estágio $writeback$ dos processadores RISC.

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* em um FPGA. A seção 5.4 apresenta uma avaliação de desempenho em comparação com o VPR [Murray et al. 2020] e o SA com múltiplos núcleos proposto em [Carvalho et al. 2020].

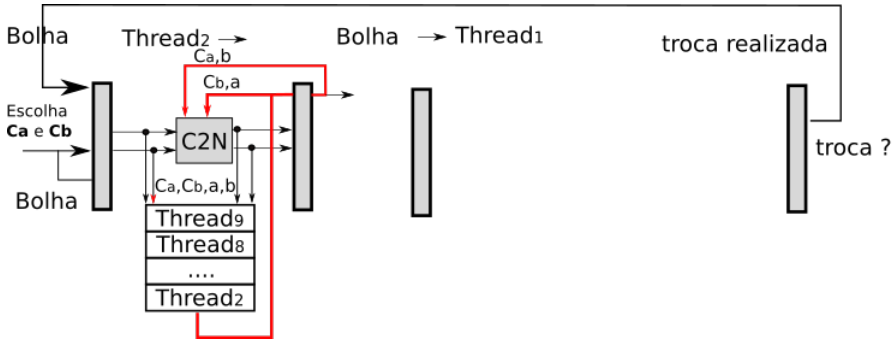

Figura 5. Como atualizar com múltiplas *threads* e redução de sinais.

5.1. Simulador em Python

Para a flexibilização do desenvolvimento, um simulador em Python foi construído para avaliar diversas configurações de troca e número de estágios. O simulador pode ser usado para depuração da execução com a exibição do conteúdo das memórias, a evolução da redução do custo por *thread* e os posicionamentos parciais. O simulador simplifica a validação do desenvolvimento de um gerador de código descrito na próxima seção. Em [Wrighton and DeHon 2003], os autores também desenvolveram um simulador para a depuração do circuito com arranjo sistólico.

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. O gerador é parametrizável com o tamanho da arquitetura alvo. Como o *pipeline* possui estrutura regular, o uso do gerador permite a exploração de várias possibilidades de projetos como a variação da profundidade do *pipeline* e/ou a complexidade dos estágios, a distribuição dos módulos de memória dentre outros aspectos que podem ser investigados.

5.3. Recursos utilizados pela Unidade de Posicionamento em *Hardware*

O código Verilog gerado foi sintetizado com a ferramenta Vivado 2020.2 para o FPGA Xilinx VU9P que possui aproximadamente 2,5 milhões LUTs, 6800 DSPs e 2160 bancos de memória BRAM com capacidade total de 7,9MB de memória [Xilinx 2020]. A tabela 1 apresenta o total de recursos utilizados para a implementação da unidade de posicionamento considerando diversos tamanhos de arquitetura do CGRA.

Os recursos necessários para implementar o posicionamento em *hardware* dependem do tamanho do CGRA alvo e do número de cópias de unidades de posicionamento. Foram avaliadas arquiteturas com 4x4, 8x8, 9x9, 10x10 e 12x12 unidades, respectivamente. A menor arquitetura de 4x4 terá 16 elementos de processamento e poderá mapear um grafo com até 16 operações. A maior arquitetura 12x12 pode mapear um grafo com até 144 operações. O tamanho da arquitetura impacta no tamanho das memórias que armazenam as soluções parciais. Para aumentar o desempenho, avaliamos o uso de 10 unidades de posicionamento executando em paralelo. Observamos a escalabilidade da solução que gasta 10x mais recursos, mas também é 10x mais rápida.

Tabela 1. FPGA Xilinx VU9P e utilização de recursos.

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

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. O sintetizador pode optar pela utilização de LUTs e/ou BRAMs em função do tamanho das memórias. Para memórias maiores, o sintetizador aloca blocos de BRAM e para menores aloca LUTs. Pode ocorrer uma fragmentação no uso das memórias, pois as BRAM possuem tamanho fixo de 32Kb. Se a unidade de posicionamento aloca 26Kb para uma determinada arquitetura, os 8Kb restantes não serão utilizadas. Entretanto, o impacto do uso de BRAMs é baixo no custo total da unidade de posicionamento, sendo que apenas 2,2% do total de BRAMs foram necessários, como podemos observar na Tabela 1.

Por fim, o único recurso demandado em maior escala são as LUTs, pois a implementação da medida de distância foi realizada com lógica combinacional. No pior caso para 10 cópias ocupou apenas 3,6% do FPGA. Esta etapa pode ser futuramente substituído por DSPs o que reduzirá significativamente o uso de LUTs.

5.4. Avaliação do Desempenho

A ferramenta VPR [Murray et al. 2020] representa o estado da arte nas duas últimas décadas para SA e é atualmente utilizada em uma parceria da Google com a Universidade de Toronto no projeto Antmicro no desenvolvimento de ferramentas de código aberto para projeto de circuitos integrados e FPGAs. As unidades de posicionamento em *hardware* foram sintetizadas, mapeadas e executadas no FPGA com 6 *threads*. Todos os experimentos mostraram o tempo de execução de 2 ciclos para cada iteração do SA.

A Tabela 2 apresenta o tempo de execução do posicionamento de duas implementações do SA em *software* [Carvalho et al. 2020, Murray et al. 2020] para um conjunto de grafos de fluxo de dados [Canesche et al. 2020], comparando com a solução em *hardware* considerando 2 ciclos por iteração do SA e uma frequência de 250MHz. A Tabela 2 mostra o tempo de execução e a qualidade da solução considerando o tamanho da fila para duas implementações em *software* do SA. A solução em *hardware* gera re-

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

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

sultados com a mesma qualidade da versão SA em *software* [Carvalho et al. 2020]. As duas primeiras colunas identificam a aplicação e o tamanho em nós do grafo de fluxo de dados, respectivamente. A coluna seguinte apresenta quantas iterações foram realizadas considerando 1.000 execuções (ou 1.000 *threads*). O uso de 1.000 execuções melhora significativamente o resultado. A coluna $T_{16threads}$ mostra o tempo de execução, tempo de cada iteração e a qualidade da solução para a implementação em *software* 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 VPR mostra o tempo de execução para 1.000 cópias do VPR com a opção *fast* ativada e a qualidade da solução em filas. 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*. Em resumo, estes resultados mostram que o SA em *software* [Carvalho et al. 2020] ou em *hardware*, proposto neste

trabalho, resolvem o problema de posicionamento melhorando a qualidade do 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. Ao comparar 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 [Carvalho et al. 2020] e VPR, respectivamente. Na comparação, foi considerada 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 passar a ser 10x mais rápida que uma unidade, passando de 31,5 vezes para 315 vezes mais rápida que a ferramenta VPR.

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 paralelização da execução. Ao comparar com a ferramenta VPR [Murray et al. 2020], a versão em *hardware* foi, em média, de 30 a 300 vezes mais rápida e melhorou a qualidade ao reduzir a quantidade de filas para o balanceamento. A solução paralela usa várias memórias e uma solução de sobreposição de escrita para maximizar o desempenho, com a execução de cada iteração do SA em 2 ciclos de relógio. Trabalhos futuros irão avaliar a integração com algoritmo de roteamento em *hardware* e também a validação para posicionamentos em nanotecnologias [Formigoni et al. 2019].

Referências

Canesche, M., Menezes, M., Carvalho, W., Torres, F. S., 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, 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)*.

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

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.

Ferreira, R., Duarte, V., Pereira, M., Carro, L., and Wong, S. (2013). A just-in-time modulo scheduling for virtual coarse-grained reconfigurable architectures. In *Int Conf on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*.

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)*.

Formigoni, R. E., Ferreira, R. S., and Nacif, J. A. M. (2019). Ropper: A placement and routing framework for field-coupled nanotechnologies. In *Symposium on Integrated Circuits and Systems Design (SBCCI)*.

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 *Int Conf 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.

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 Int Symp on Circuits and Systems (ISCAS)*.

Walker, M. J. and Anderson, J. H. (2019). Generic connectivity-based CGRA mapping via integer linear programming. In *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*.